August 1989

Report No. STAN-CS-89-1282

Thesis

AUTOMATIC RUNTIME CONSISTENCY CHECKING AND DEBUGGING OF FORMALLY SPECIFIED PROGRAMS

by

Sriram Sankar

Department of Computer Science

Stanford University

Stanford, California 94305

unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION

1b RESTRICTIVE MARKINGS

2a SECURITY CLASSIFICATION AUTHORITY

2b DECLASSIFICATION / DOWNGRADING SCHEDULE

3 DISTRIBUTION / AVAILABILITY OF REPORT
Unclassified: Distribution Unlimited

4 PERFORMING ORGANIZATION REPORT NUMBER(S)
STAN-CS-89-1282

5 MONITORING ORGANIZATION REPORT NUMBER(S)

6a NAME OF PERFORMING ORGANIZATION
Computer Science Dept.

6b OFFICE SYMBOL (If applicable)

7a NAME OF MONITORING ORGANIZATION

6c ADDRESS (City, State, and ZIP Code)
Stanford University
Stanford, CA 94305

7b ADDRESS (City, State, and ZIP Code)

8a NAME OF FUNDING / SPONSORING ORGANIZATION
DARPA

8b OFFICE SYMBOL (If applicable)

9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
N00039-84-C-0211

8c ADDRESS (City, State, and ZIP Code)
Arlington, VA

10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
Automatic Runtime Consistency Checking and Debugging of Formally Specified Programs

12 PERSONAL AUTHOR(S)
Sriram Sankar

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
thesis	FROM _____ TO _____	1989-August	210

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

see other side...

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS

21 ABSTRACT SECURITY CLASSIFICATION

22a NAME OF RESPONSIBLE INDIVIDUAL
Luckham

22b TELEPHONE (Include Area Code)
415-723-2273

22c OFFICE SYMBOL

DD Form 1473, JUN 86 Previous editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603

AUTOMATIC RUNTIME CONSISTENCY CHECKING AND DEBUGGING OF FORMALLY SPECIFIED PROGRAMS

Sriram Sankar
Stanford University, 1989

Dissertation Advisor: Prof. David C. Luckham

Abstract: This thesis studies an approach to automate the process of deciding whether a program is performing correctly, and if not, to discover the probable cause of the problem. It assumes that the intended behavior of the program is specified in some formal, high-level specification language. It studies how one can check automatically at runtime whether the program is running consistently with its specification, and if not, how inconsistencies can be automatically detected and diagnosed. A methodology of using this checking methodology for debugging formally specified programs is then presented.

The consistency checking methodology depends on the particular specification language constructs used. In this thesis, two categories of constructs are studied: *(1) generalized assertions* and *(2) algebraic specifications.*

Generalized assertions contain boolean expressions that must be satisfied within a specified region in the underlying program. *Checking functions* are generated which test for the truth of these boolean expressions. Diagnostic messages are given and a debugger is invoked if there is a violation. Checking functions are called from locations in the program where the specification may have changed value.

For the purpose of this thesis, algebraic specifications are considered to be *equations* whose terms comprise abstract data type operations. Algebraic specification checking involves monitoring the execution of the abstract data type operations. Based on this monitoring and the algebraic specifications, a theorem prover generates invariants that the program must satisfy. If the program does not satisfy these invariants, diagnostic messages are given and a debugger is invoked. The theorem prover has to be specialized so that it operates efficiently in the context of algebraic specification checking. Methodologies to achieve this using incremental techniques are presented in this thesis.

Based on these ideas, a working system has been built for automatic runtime consistency checking of Ada programs with spcifications written in Anna. Experiments with this system has led to the development of a methodology of debugging programs based on formal specifications.

AUTOMATIC RUNTIME CONSISTENCY CHECKING AND DEBUGGING OF FORMALLY SPECIFIED PROGRAMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Sriram Sankar
August 1989

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

David C. Luckham
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Mark A. Linton

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Natarajan Shankar

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

This thesis studies an approach to automate the process of deciding whether a program is performing correctly, and if not, to discover the probable cause of the problem. It assumes that the intended behavior of the program is specified in some formal, high-level specification language. It studies how one can check automatically at runtime whether the program is running consistently with its specification, and if not, how inconsistencies can be automatically detected and diagnosed. A methodology of using this checking methodology for debugging formally specified programs is then presented.

The consistency checking methodology depends on the particular specification language constructs used. In this thesis, two categories of constructs are studied: *(1) generalized assertions* and *(2) algebraic specifications.*

Generalized assertions contain boolean expressions that must be satisfied within a specified region in the underlying program. *Checking functions* are generated which test for the truth of these boolean expressions. Diagnostic messages are given and a debugger is invoked if there is a violation. Checking functions are called from locations in the program where the specification may have changed value.

For the purpose of this thesis, algebraic specifications are considered to be *equations* whose terms comprise abstract data type operations. Algebraic specification checking involves monitoring the execution of the abstract data type operations. Based on this monitoring and the algebraic specifications, a theorem prover generates invariants that the program must satisfy. If the program does not satisfy these invariants, diagnostic messages are given and a debugger is invoked. The theorem prover has to be specialized so that it operates efficiently in the context of algebraic specification checking. Methodologies to achieve this using incremental techniques are presented in this thesis.

Based on these ideas, a working system has been built for automatic runtime consistency checking of Ada programs with specifications written in Anna. Experiments with this system has led to the development of a methodology of debugging programs based on formal specifications.

Acknowledgements

I am deeply indebted to my parents for their motivation and support. If not for them, I would never have embarked on this ambitious project. My wife, Uma, has cheerfully endured with me the problems and tensions of my life at Stanford. For this and her constant emotional support I am extremely grateful.

My advisor, David Luckham, has been a great source of inspiration. I am grateful for his advice, motivation and support. I also thank my other committee members, Mark Linton and Natarajan Shankar, for their comments and suggestions to improve this thesis.

A large number of people have contributed in one way or the other to this research. I am thankful to Doug Bryan, Neel Madhav, Walter Mann, Sigurd Meldal, Geoff Mendal, Randall Neff, Arun Swami and Friedrich vonHenke for their contributions through my interactions with them.

Special thanks are due to David Rosenblum who participated in the initial design and development of the generalized assertion checking methodology and the concurrent checking methodology. The debugging experiments performed by Shuzo Takahashi and David Luckham formed the basis for Chapter 5.

A lot of support software has been developed within this research group. The Anna semantics checker was developed by Geoff Mendal. Manas Mandal implemented concurrent checking of some Anna constructs. Pilot versions of an overload resolution system and the axiom preprocessor were implemented by Rob Chang and John Sebes respectively.

The Anna Consistency Checking System has been comprehensively tested on large programs by the DADAISM group, especially by John Kenney. This has helped in creating a more sturdy system capable of handling large programs.

Finally, I am thankful to Rosemary Brock for her efficient handling of various administrative details and for helping out with my initial typesetting problems.

This research was supported by the Defense Advanced Research Projects Agency under contract N00039-84-C-0211.

Contents

Abstract iv

Acknowledgements v

1 Introduction 1

1.1 Contributions of this Thesis 2

1.1.1 The Anna Consistency Checking System 3

1.1.2 Examples 5

1.2 Organization of this Thesis 8

1.3 Related Work 10

1.3.1 Instrumentation 10

1.3.2 Testing 12

1.3.3 Static Analysis 13

1.3.4 Symbolic Analysis 14

2 Consistency Checking Principles 15

2.1 The Specification Language 15

2.2 Automatic Runtime Consistency Checking 16

3 Generalized Assertions Checking 18

3.1 Overview 18

3.2 Checking Functions 19

3.3 Transformation to Basic Annotations 25

3.3.1 Simple Statement Annotations 25

3.3.2 Compound Statement Annotations 27

3.3.3 Subprogram Annotations 28

3.4 Transformation of Anna Expressions 29
3.4.1 Stage 1: Implication Operators and Anna Membership Operators . . 29
3.4.2 Stage 2: Conditional Expressions 31
3.4.3 Stage 3: Anna Relational Expressions 37
3.4.4 Stage 4: Initial Names and Expressions 38
3.4.5 Stage 5: Renaming of Objects 38
3.4.6 Quantified and State Expressions 39
3.5 Generation of Checking Code 39
3.5.1 Checking Functions of Subtype Annotations 39
3.5.2 Checking Functions of Object Annotations 40
3.5.3 Checking Functions of Result Annotations 45
3.5.4 Calls to Checking Functions 46
3.5.5 Transformation of Exception Annotations 55
3.5.6 Transformation of Out Annotations 60
3.6 Concurrent Checking of Generalized Assertions 64

4 Algebraic Specification Checking **65**
4.1 Abstract Data Types and Algebraic Specifications 65
4.2 Algebraic Specification Checking 66
4.3 Complexity of the General Problem 72
4.4 Thue Systems 74
4.4.1 An Overview of Proof Theory in Equational Logic 74
4.4.2 Terminology, Definitions and Lemmas 75
4.4.3 Matching Abstract Data Types to Thue Systems 78
4.5 The Chromatic Theorem Prover 82
4.5.1 Some Terminology and Definitions 83
4.5.2 Constructing Rewrite Rules from Equations 83
4.5.3 The Neighbor Set 84
4.5.4 The Theorem Proving Step 86
4.5.5 Termination 86
4.6 Incremental Execution of the Algorithm 86
4.6.1 Detailed Description of the Algorithm 88
4.6.2 Comparison with the Chromatic Theorem Prover 91

4.6.3 A Specialized Two-Color Algorithm 94
4.7 Capabilities of the Algorithm 96
4.8 Miscellaneous Topics 100
4.8.1 Going Outside the Subset 100
4.8.2 Undefinedness of Expressions 101
4.8.3 Concurrent Algebraic Specification Checking 102

5 Debugging Formally Specified Programs **103**
5.1 The Anna Debugger 103
5.2 Operation Sequences and Structural Levels 106
5.3 Assumptions about Specifications 108
5.4 Two-Dimensional Pinpointing 108
5.5 Ada Packages 112
5.6 An Illustrative Debugging Session 114
5.6.1 The QUEUE_MANAGER Package 114
5.6.2 The Debugging Session 118

6 Conclusions **131**

A An Overview of Ada **134**
A.1 Ada Programs 134
A.2 Subprograms 135
A.3 Packages 137
A.4 Exceptions 139
A.5 Declarations 141
A.5.1 Type and Subtype Declarations 141
A.5.2 Object Declarations 143
A.5.3 Renaming Declarations 144
A.6 Statements 144
A.6.1 Null Statements 144
A.6.2 Assignment Statements 144
A.6.3 If Statements 145
A.6.4 Case Statements 145
A.6.5 Loop and Exit Statements 146

A.6.6 Block Statements 147
A.6.7 Goto Statements 147
A.7 Names and Expressions 148

B An Overview of Anna **152**
B.1 Anna Formal Comments 153
B.1.1 Virtual Ada Text 153
B.1.2 Annotations 154
B.2 Anna Expressions 155
B.2.1 Quantified Expressions 155
B.2.2 Conditional Expressions 155
B.2.3 State Expressions 156
B.2.4 Initial Expressions 157
B.2.5 Anna Operators 158
B.3 Annotations 158
B.3.1 Object Annotations 158
B.3.2 Subtype Annotations 159
B.3.3 Statement Annotations 160
B.3.4 Subprogram Annotations and Result Annotations 161
B.3.5 Axiomatic Annotations 162
B.3.6 Context Annotations 163
B.3.7 Exception Annotations 163

C Installation Manual and User Guide **165**
C.1 Introduction 165
C.2 Installing the Anna System 166
C.2.1 Setting Up the Machine Dependent Parameters 166
C.2.2 Compiling the Anna Consistency Checking System 168
C.2.3 Setting Up the Predefined Environment 169
C.3 Non-Standard Anna Features 169
C.3.1 Annotation Names 169
C.3.2 Anna Pragmas 170
C.4 Transforming Anna Programs 172
C.4.1 Creating the Self-Checking Executable 173

C.4.2 The Anna Debugger 174
C.5 Subset Restrictions: The Ada Reference Manual 175
C.5.1 Introduction 176
C.5.2 Lexical Elements 176
C.5.3 Declarations and Types 176
C.5.4 Names and Expressions 177
C.5.5 Statements 179
C.5.6 Subprograms 179
C.5.7 Packages 179
C.5.8 Visibility Rules 180
C.5.9 Tasks 181
C.5.10 Program Structure and Compilation Issues 181
C.5.11 Exceptions 182
C.5.12 Generic Units 182
C.5.13 Representation Clauses/Implementation-Dependent Features 182
C.5.14 Input-Output 182
C.6 Subset Restrictions: The Anna Reference Manual 182
C.6.1 Basic Anna Concepts 182
C.6.2 Lexical Elements 182
C.6.3 Annotations of Declarations and Types 182
C.6.4 Names and Expressions in Annotations 183
C.6.5 Statement Annotations 184
C.6.6 Annotation of Subprograms 185
C.6.7 Packages 185
C.6.8 Visibility Rules in Annotations 186
C.6.9 Tasks 186
C.6.10 Program Structure 186
C.6.11 Exception Annotations 186
C.6.12 Annotation of Generic Units 186
C.6.13 Annotation of Implementation-Dependent Features 186

Bibliography **187**

List of Figures

1.1 The Anna Consistency Checking System 4
1.2 Thesis Reading Guide 11

3.1 Transformation of Annotations to Checking Functions 23
3.2 Transformation of Simple Statement Annotations 26
3.3 Transformation of Compound Statement Annotations 27
3.4 Transformation of Subprogram Annotations 28
3.5 Transformation of Anna Expressions 30

4.1 The Incremental Chromatic Theorem Prover 87
4.2 The Four Graphs of Example 4.7 90
4.3 The Five Graphs of Example 4.8 91
4.4 The Five Graphs of Example 4.9 96
4.5 Order 2 Algorithm Graphs of Example 4.10 97
4.6 Order 2.5 Algorithm Graphs of Example 4.10 97

5.1 The Generalized Assertion Checking/Debugging Subsystem 104
5.2 The Algebraic Specification Checking/Debugging Subsystem 105
5.3 A Typical Anna Debugger Screen Layout 105
5.4 Operation Sequences 106
5.5 Structural Levels within the QUEUE Package 113
5.6 Result of Interaction 1 119
5.7 Result of Interaction 2 122
5.8 Result of Interaction 3 124
5.9 Result of Interaction 4 126
5.10 Result of Interaction 5 128

5.11 Result of Interaction 6 129
5.12 The Region of Suspicion at Each Interaction 130

Chapter 1

Introduction

At the start of the programming task, the programmer is supplied with a *specification* of the problem. The specification may be as formal as a document which details the intended behavior of the program in all possible circumstances or it may be as informal as a few instances of what the program is intended to do. In practice, the programmer has available several sources of information which comprise the specification. These may include a formal specification document, a working prototype, instances of program behavior, and *a priori* knowledge about similar software. All of these sources contribute to the programmer's understanding of the task. Formal specifications are usually written in some *specification language.* Specification languages provide formal, high-level constructs to describe various properties of programs. An overview of some formal specification techniques is given in [67].

Working from this specification, the programmer develops the program. The validation of the program lies in the comparison of the program with the specification of intended behavior. If the specification is completely formal, then validation can be performed by providing a mathematical proof that compares the program with the formal specification. This process is termed *program verification.* An overview of program verification is given in [68]. However, specifications are hardly ever formalized completely, and even when they are, they can still contain errors. Even if the existence of a complete and correct specification is assumed, it is still very difficult to provide the necessary mathematical proof.

It is much easier to perform the validation process for a *particular* input data. In this situation it is typically possible to obtain the intended behavior of the program by hand calculation, textbook requirements or by the application of estimates obtained from simulations. If the specification is completely formal, it may even be possible to automatically

determine the intended behavior from the specification. Repeating the process of comparing the program with the intended behavior on many different inputs can increase the level of confidence on this program. This process is termed *program testing.* Program testing can also test specifications. If the intended behavior of the program determined from the specifications is different from the expected behavior, there is a problem with the specifications.

There are two parts to program testing. First, there is the problem of *test-data selection.* Since any test-data will necessarily be a very small sample of all the possible input data, test-data should be selected in such a way that successful execution of a program on these test-data give us a reasonable amount of confidence in the correctness of the program for all possible input data. At the same time, the test-data should be such that redundancy in the testing process is minimized. The second part to program testing is to run the program on each test-data to determine whether or not the program implements the intended behavior for this test-data. An overview of program testing is given in Section 1.3.

The details of determining whether or not the program implements the intended behavior for a particular test-data is usually not dealt with in program testing theory. Rather an *oracle* is assumed to exist. This oracle can judge for any specific test-data, whether or not the program implements its intended behavior. The idealization of the oracle is essential for software testing. Various testing strategies have handled the oracle problem in different ways.

1.1 Contributions of this Thesis

A typical scenario in which the oracle works consists of a specification, a program that has been designed to be consistent with the specification and some test-data. The specification is written formally (at least partially) in some specification language. The program is usually referred to as the *underlying program.* Hence, at least a part of the work done by the oracle is in comparing the program with the formal specifications during execution on the input data[1]. This thesis provides a methodology to automate *this aspect* of the oracle's work. This methodology is referred to in this thesis as *automatic runtime consistency checking.* This thesis does not, however, address the problem of test-data selection.

Obviously, the complexity of the methodology depends on the kinds of constructs available in the specification language. In this thesis, two different categories of specification

[1]The other part of the oracle's work may be in comparing the program with respect to informal specifications.

constructs are assumed. The first category includes constructs that are generalizations of *assertions*. Consistency checking for this category of specification constructs is performed by creating functions corresponding to these constructs and instrumenting the program with calls to these functions. These functions are referred to as *checking functions*.

The second category of specification constructs is the *algebraic specification*. Algebraic specifications describe abstract data types. For the purpose of this thesis, algebraic specifications are defined as equations whose terms comprise the abstract data type operations and variables that are universally quantified over the domain of the abstract data type. Consistency checking for this category of specification constructs is performed with the help of a theorem prover, which performs proofs on sequences of operations executed by the underlying program. Unless properly performed, this form of consistency checking can consume an unacceptably large overhead in time. To solve this problem, an incremental theorem prover has been designed that works on a small but useful subset of abstract data types and algebraic specifications. This theorem prover is referred to as the *Chromatic Theorem Prover*.

1.1.1 The Anna Consistency Checking System

Everything described in this thesis has been fully implemented. The programming language used is Ada (see Appendix A) and the specification language used is Anna (see Appendix B). The user-interface of the system includes a specialized *Anna Debugger* designed to handle situations where the underlying program becomes inconsistent with the specification. This system is known as the *Anna Consistency Checking System* and has been completely implemented in Ada at Stanford University. Various versions of the system have been ported and are being used at many locations around the world. An installation manual and user's guide of the latest release of this system is given in Appendix C. A block diagram of the system is shown in Figure 1.1.

The Anna specifications are converted to *checking-code* and this is instrumented into the underlying Ada program. This process is performed by a tool called the *Anna Transformer*. The resulting Ada program is now compiled using a standard Ada compiler and then linked and loaded together with the Anna Debugger and the Chromatic Theorem Prover. The result is a self-checking executable. The instrumented Ada program makes calls to the checking code every time a specification may potentially be violated. The checking code determines whether or not a violation has taken place. When checking with respect to

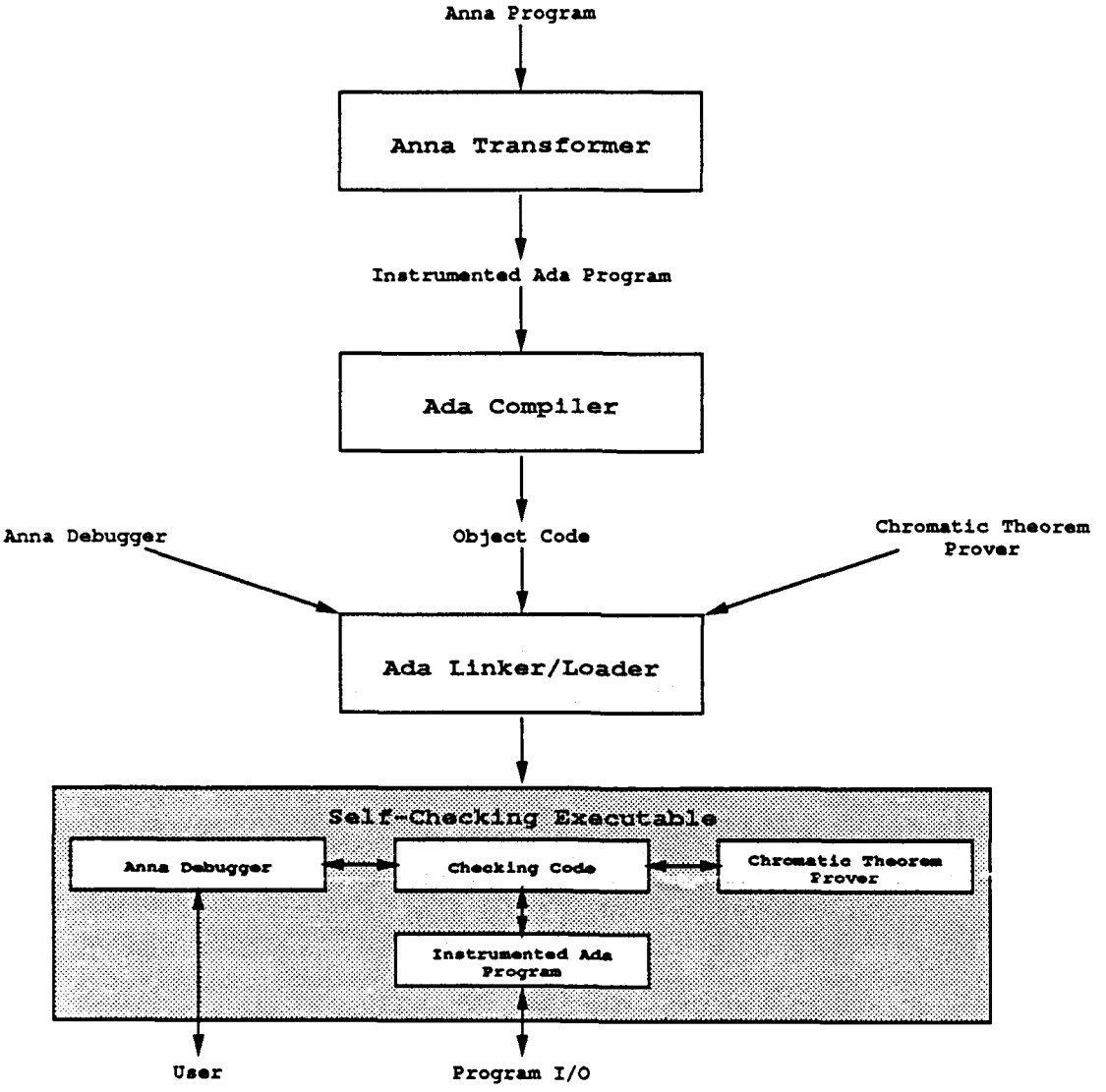


Figure 1.1: The Anna Consistency Checking System

algebraic specifications, the checking code makes calls to the Chromatic Theorem Prover. If the checking code determines that a violation has taken place, control is transferred to the Anna Debugger. The user can then interact with the Anna Debugger to get more information on the nature of the inconsistency. The structure of the self-checking executable is also shown in Figure 1.1. The Chromatic Theorem Prover is described in Chapter 4, while the Anna Debugger is described in Chapter 5.

The Anna Transformer converts the Anna program to a corresponding parse tree using a table driven parser. The parse tree data structure and operations on this data structure are encapsulated together to form the *AST (Abstract Syntax Tree) package*. The AST package is based on *DIANA* [22]. DIANA is an intermediate tree representation for Ada programs. The AST package extends the DIANA concepts to handle Anna constructs. The Anna Transformer then transforms the Anna parse tree to an Ada parse tree by changing the Anna specifications to Ada checking code. The Ada parse tree is then converted to an Ada program using a reverse parser[2]. The Anna Transformer also includes a semantic analyzer[3] which checks the original Anna program for static semantic correctness. This notion of correctness is what is determined by compiler front-ends and not the notion of correctness determined by program verification tools.

The complete transformation process includes the introduction of new constructs like variable declarations and renaming declarations; transformation of Anna expressions to equivalent Ada expressions; the transformation of annotations to more basic annotations; and the generation of checking code. For a complete description of the transformation process, please refer to [108]. Relevant details of this process will be illustrated in this thesis.

1.1.2 Examples

Two examples of Anna code fragments (Ada code with Anna specifications) are shown below. These examples illustrate runtime consistency checking and issues involved in its automation.

Example 1.1: This example illustrates consistency checking of Anna *subtype annotations*. It contains a subtype annotation which constrains the Ada subtype EVEN to take on only

[2]Note that the Ada parse tree produced by the Anna Transformer can be fed directly to any Ada compiler that uses DIANA as its intermediate representation.

[3]The semantic analyzer was developed by Geoff Mendal.

those values that are divisible by 2.

```
declare
   subtype EVEN is INTEGER;
   --| where X:EVEN => X mod 2 = 0;
   E:EVEN;
begin
   E := 4;
   E := E+1;
end;
```

The first value assigned to E, namely 4, satisfies the subtype annotation. However, after the second assignment, the value of E will be 5 and this does not satisfy the subtype annotation. To implement consistency checking with respect to subtype annotations, a test is inserted for each of the statements that modify the variable E. During execution, the program passes the test corresponding to the first assignment statement, but the test corresponding to the second assignment statement detects an inconsistency.

Example 1.2: In this example, the problem of checking the consistency of abstract data type operations with respect to algebraic specifications is illustrated. The abstract data type shown below represents *deques*. The operations are CREATE, LEFT_PUSH, LEFT_POP, LEFT_TOP, RIGHT_PUSH, RIGHT_POP and RIGHT_TOP.

```
package DEQUE_PACKAGE is
   type DEQUE is ...;
   function CREATE return DEQUE;
   function LEFT_PUSH(D:DEQUE;E:ELEMENT) return DEQUE;
   function LEFT_POP(D:DEQUE) return DEQUE;
   function LEFT_TOP(D:DEQUE) return ELEMENT;
   function RIGHT_PUSH(D:DEQUE;E:ELEMENT) return DEQUE;
   function RIGHT_POP(D:DEQUE) return DEQUE;
   function RIGHT_TOP(D:DEQUE) return ELEMENT;
```

```
--| axiom
--|     for all D:DEQUE;E:ELEMENT =>
--|         LEFT_POP(LEFT_PUSH(D,E)) = D,
--|         RIGHT_POP(RIGHT_PUSH(D,E)) = D,
--|         LEFT_POP(RIGHT_PUSH(D,E)) = RIGHT_PUSH(LEFT_POP(D),E),
--|         RIGHT_POP(LEFT_PUSH(D,E)) = LEFT_PUSH(RIGHT_POP(D),E),
--|         LEFT_TOP(LEFT_PUSH(D,E)) = E,
--|         RIGHT_TOP(RIGHT_PUSH(D,E)) = E;
end DEQUE_PACKAGE;
```

This example contains six Anna package axioms. These axioms form the algebraic specification of the deque package. The deque package is expected to satisfy these axioms; otherwise it is inconsistent with these axioms. Note however that this is not a complete specification of deques. For example, the special cases of empty and full deques have been ignored[4]. There are also some other facts about non-empty deques that cannot be proved from these axioms. The following is one such fact:

$$\text{LEFT\_PUSH}(\text{RIGHT\_PUSH}(D, E_1), E_2) = \text{RIGHT\_PUSH}(\text{LEFT\_PUSH}(D, E_2), E_1)$$

where D, E_1 and E_2 are assumed to be universally quantified over the appropriate domains. The discussion that follows assumes that all facts that the deque package *must* satisfy can be derived from just the six axioms mentioned above. Any intuitive facts that cannot be derived from these axioms (like the one mentioned above) can be either true or false of the deque package. The deque package will still be consistent with the axioms.

The following is a sequence of statements that invokes the deque package at runtime:

```
D0 := CREATE;
D1 := LEFT_PUSH(D0, E0);
E1 := LEFT_TOP(D1);
D2 := LEFT_POP(D1);
```

The axioms require that after successful execution of the above statements, E_1 has to be equal to E_0 and D_2 has to be equal to D_0. Therefore, runtime consistency checking requires

[4]Note that undefined expressions do not violate Anna package axioms. Hence, though the third and fourth equations are undefined (typically) for the empty deque, this does not cause the package to be inconsistent with the specification.

that checks be made at runtime to ensure that these equalities do indeed hold. Hence, in this example, after the assignment to E_1, a check is made to ensure that E_1 is equal to E_0 and after the assignment to D_2, a check is made to ensure that D_2 is equal to D_0. Note however that it is not possible in general to *insert* these checks inline when the program is compiled. This is because the performance of the check is based on the execution of a certain sequence of operations of the deque package, rather than just one particular operation. It is not always possible at compile time to determine when such sequences will be executed. If the second statement in this example was within an *if* statement, then the above-mentioned checks would have to be made depending on whether or not the condition of the *if* statement was true. Also note that the sequences of operations after which the tests need to be performed are not always as simple as in the above example. For example, after execution of the following sequence of statements:

D_3 := CREATE;
D_4 := LEFT_PUSH(D_3, E_2);
D_5 := RIGHT_PUSH(D_4, E_3);
D_6 := LEFT_POP(D_5);
D_7 := RIGHT_POP(D_6);

the following condition has to hold:

$$D_7 = D_3$$

The problem of inserting checks is solved as follows: After each execution of a package operation, a message is sent to the Chromatic Theorem Prover with the necessary information. The theorem prover maintains a history of the package operations already performed. Based on this information, the theorem prover decides whether or not a check needs to be performed. If a check has to be performed, the theorem prover sends the appropriate message to the checking code.

1.2 Organization of this Thesis

This thesis consists of six chapters and a set of supporting appendices. The aim of the chapters is to concisely describe the contributions of the thesis. The appendices, on the other hand, provide useful background information.

Chapter 2 is a short extension of this chapter. It discusses the principles of consistency checking. It further describes the problem of consistency checking with respect to formal specifications in more detail. It introduces the two different approaches to consistency checking and sets the stage for the more detailed description of the consistency checking methodology.

Chapter 3 describes the checking function methodology. It explains the concept of checking functions, their use and advantages. A detailed discussion of the transformations involved for the checking of generalized assertions follows. Many examples in Ada/Anna are given. Though all transformations described in this chapter are specific to Ada/Anna programs, they can be applied to other languages with minor modifications. Some of the transformations discussed in this chapter—like the transformation of Anna expressions—also apply to algebraic specifications.

Chapter 4 describes the consistency checking of abstract data type implementations with respect to algebraic specifications. First, all the transformations that need to be performed on Ada abstract data types and Anna algebraic specifications are described. Then the general problem is shown to be undecidable (based on previous work), and hence only partial checking methodologies are possible. The Chromatic Theorem Prover is discussed in this chapter. This chapter also explains how theorem proving operations can be performed incrementally, and concludes with describing the capabilities of this prover.

Chapter 5 describes how one can use the Anna Consistency Checking System to debug programs based on formal specifications. This chapter concentrates on the problem of debugging Ada packages. The methodologies developed in this chapter can be easily generalized to apply beyond Ada packages to more complex programs. A session where a QUEUE_MANAGER package is debugged illustrates the application of the debugging methodology. In addition, this example also illustrates how the Anna Consistency Checking System is used.

Chapter 6 concludes this thesis. The contributions of this thesis and possible future work are discussed.

Appendix A gives a reasonably comprehensive introduction to Ada, and for the reader not familiar with Ada, this appendix should be sufficient for the purposes of understanding this thesis. In fact, this appendix can also be used effectively as a self-contained introductory text to Ada.

Appendix B similarly gives a reasonably comprehensive introduction to Anna.

Appendix C is an installation manual and user guide for the Anna Consistency Checking System. This appendix explains in detail how this system can be set up on a machine, and how it can be used. It also gives a comprehensive description of the limitations of the system.

Figure 1.2 is a reading guide for this thesis. It may be used to determine what needs to be read before the section of interest is read. In this figure, a solid arrow from A to B means that A must be read before reading B. A dashed arrow from A to B means that it will be useful, though not essential to read A before reading B. The sections within the dotted ovals can be omitted during the first reading of the thesis. It may be useful to read Appendix A and Appendix B before reading Chapters 3, 4 and 5. Section C.3.1 must be read before reading Chapter 5.

1.3 Related Work

1.3.1 Instrumentation

Enhancing a program with additional code has been used for a variety of purposes for a very long time. This process is called *instrumentation*. The earliest uses of instrumentation have been to gather statistics about the program. Examples of such statistics are statement execution counts and branch of control counts. Some of the earliest work in instrumentation has been by Estrin et al. [21,106] and Knuth et al. [51,58]. Around the mid-70's, the idea of using instrumentation for consistency checking began to emerge. Some work in this area has been done by Yau and Cheung [123] who proposed that introducing redundancy in a program could create self-checking programs. Stucki [113] was one of the first to implement automatic runtime consistency checking of a program with respect to assertions. Since then many systems have been built where assertions have been compiled into runtime checking code. There are other related uses of redundancy for self-checking purposes. Some obvious examples are constraint checking that programming languages like Pascal and Ada provide, and hardware error detection and correction. Another example is data structure error detection and correction [115]. Bird and Munoz [8] have designed a scheme to generate self-checking test-data for compilers. The idea here is to introduce extra code in the test-data (which is itself a program) which automatically checks the correctness of the compiler during execution of the compiled program (the test-data). Lu [69] describes a method for creating real-time self-checking software. A separate processor termed the *Watchdog Processor* is

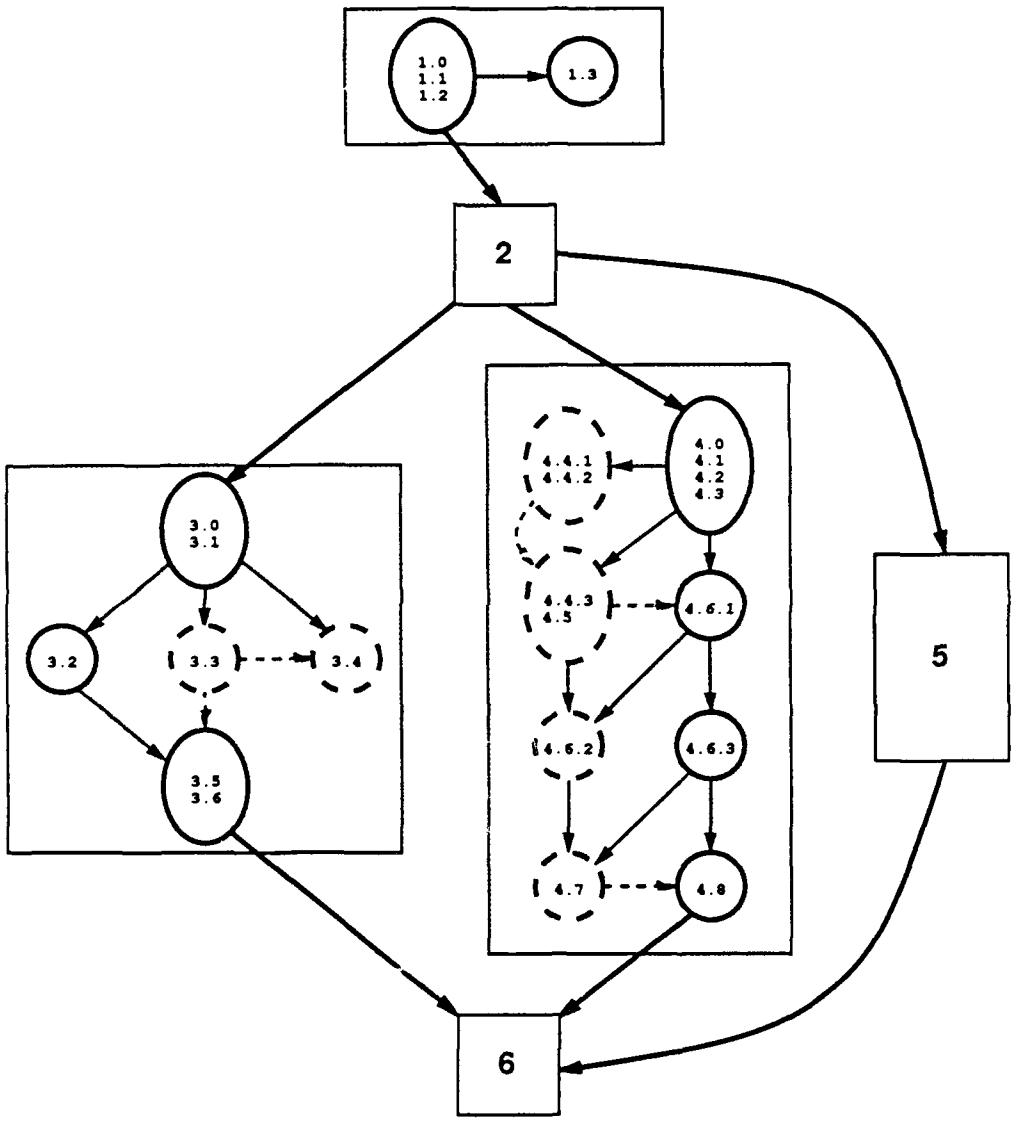


Figure 1.2: Thesis Reading Guide

used to monitor the running of the underlying program.

1.3.2 Testing

Test-Data Generation

A large number of tools have been designed for test-data generation since the early-70's. The emphasis is to generate test-data that exercises as much of the program code as practically possible. Some approaches have been to generate test-data that force every program statement to be executed, while others force every edge in the program's flowchart to be traversed. Goodenough and Gerhart [29] attempt to define a theoretically sound, but practical definition of what constitutes an adequate test. The idea is to divide the test-data into a finite number of equivalence classes where testing on a representative of an equivalence class will, by induction, test the entire class. A useful technique for test-data generation is symbolic execution of the program [11,55]. Symbolic execution can be performed in a forward traversal or a backward traversal of the program paths. During these traversals, various constraints are established which are then used to generate the test-data.

All these approaches to test-data generation fall under the general category of *white-box testing*. In white-box testing, the structure of the program is examined and test-data are derived from the program's logic. The other category is *black-box testing*. This is also known as *functional testing*. In this case, the internal structure and behavior of the program is not considered. The objective is to find out solely when the input-output behavior of the program does not agree with its specifications. In this approach, test-data are constructed from its specifications [1,89].

Program mutation [18,64] is a technique for the measurement of test-data adequacy. In mutation testing, test-data is applied to the program being tested and its *mutants* (programs that contain one or more likely errors). The test-data is considered adequate (with respect to the mutants) if the test-data reveals the errors in each of the mutants. If a program runs successfully on this test-data, it means that the program does not contain the kinds of errors present in the mutants.

Debugging

Thirty years ago, program debugging typically involved inspection of the core dump of a program after it reached a probably inconsistent state. Since then a lot of refinements

have taken place. The next stage was to provide program tracing and an user-interface from where a programmer could insert breakpoints and inspect values of variables in the program. Then came the concept of *structured programming* [17]—a systematic approach to programming that would reduce the possibility of errors. Structured programming also enhances the programmers' understanding of their programs, thus making the debugging process easier. Other advances in debugging involve the use of assertions to define breakpoints (this is essentially an application of self-checking programs). Some heuristic approaches to debugging have also been developed, for example, the *Programmer's Apprentice* [102,110]. Here the system is supplied with a *plan* and it infers the existence and type of bugs in the program by comparing the program with the plan. Shapiro [111] has developed a debugging system which diagnoses and attempts to correct bugs based on information accumulated in a database. This information is based on previous runs of the program and also input from the user. In yet another approach, *algebraic program testing* [49], program correctness may be thought of as a program equivalence problem. Since the equivalence of two programs written in a powerful enough programming language is undecidable, this approach requires programs to lie in some restricted class for which testing on a small set of test-data is sufficient to prove program equivalence. Algebraic program testing provides a theoretically sound way of determining program correctness for restricted classes of programs. More closely related to the Anna Consistency Checking System are the *TSL Monitor* [43,41] and the *VAL Transformer* [7]. These are currently being developed at Stanford for runtime consistency checking against the specification languages TSL [44] and VAL.

1.3.3 Static Analysis

In static analysis, the requirements and design documents and the code are analyzed, either manually or automatically, without actually executing the code. Only limited analysis of programs containing array references, pointer variables, and other dynamic constructs is possible using this technique. Experimental evaluation of code inspection and code walkthroughs has found these static analysis techniques to be very effective in finding from 30% to 70% of the logic design and coding errors in a typical program.

Program verification is a static analysis technique. It involves demonstrating the consistency between the program and its specification. The most frequently used verification method is based on inductive assertions [23,92]. Assertions about the program are placed

in the text of the program. The goal of a successful verification is to show that each assertion is true every time the program control passes the location of the assertion. There is a lot of tedious work involved in program verification. To assist in this, many program verification systems have been developed. Some of these verification systems are: King [56]; Stanford [71,117,114]; and Boyer-Moore [12].

Another static analysis technique is *automatic program generation.* This involves the generation of the program automatically from the specification. The generation process is such that the program is guaranteed to be consistent with the specification. Efforts in automatic program generation include EIFFEL [88,87], REFINE [30,31,32,33] and PROSPECTRA [61].

1.3.4 Symbolic Analysis

During symbolic testing, input data and program variable values are given symbolic values. The possible executions of a program are characterized by an execution tree. The execution is performed by a system called a symbolic evaluator whose major components are a symbolic evaluator and an expression simplifier.

Symbolic execution can be used to prove the correctness of a program. A program may be thought of as a finite set of assertion-to-assertion paths. If each path is shown to be correct, then the program is correct. When a program contains loops, the execution tree may contain infinite branches. Two possible methods for analyzing loops are informal inductive assertions and recurrence relations describing the behavior of each variable affected by the loop.

The use of specification languages for the formal specification and documentation of software brings with it a new set of problems: errors may occur in the specification itself. Whether using such specifications as a guide to implementation of a software package, or as an aid in debugging various prototype implementations, the task is greatly simplified by the assumption that the specifications are correct. *Specification analysis* tools [73,82,93] are being developed at Stanford. These tools symbolically execute Ada packages based on the specifications. This execution results in the updating of the symbolic state of the program. Queries regarding this state can be made by the programmer. Responses to these queries by the specification analysis tools aid the programmer in debugging specifications.

Chapter 2

Consistency Checking Principles

In Chapter 1, the problem of consistency checking of a program with respect to its specification was illustrated with examples. This problem will be characterized more precisely in this chapter. This chapter also provides some background on implementation methodologies for consistency checking. Details of these methodologies will be discussed in the subsequent chapters.

2.1 The Specification Language

The basic feature of runtime consistency checking is that it is performed *while* the program is executing. This is unlike other activities like program verification and automatic program generation which are performed *before* the program is executed. Hence the specification language must define precisely the constraints it places on various intermediate program states during its execution for runtime consistency checking to make any sense. Anna defines the concept of an *observable state* (see Appendix B). Anna specifications constrain the underlying Ada program only at these observable states. The methodologies discussed in this thesis will be applicable to any specification language with precise definitions of the constraints it imposes during the execution of the underlying program.

As has already been mentioned in Chapter 1, this thesis deals with two categories of specification language constructs—generalizations of assertions; and algebraic specifications. In Anna, object annotations, subtype annotations, statement annotations, subprogram annotations, result annotations and exception annotations fall into the first category. Anna axiomatic annotations provide the capability of first-order logic to specify programs. Hence

algebraic specifications can be written out as axiomatic annotations. Axiomatic annotations that are not algebraic specifications will not be dealt with in this thesis.

2.2 Automatic Runtime Consistency Checking

A lot of analysis and testing can be performed on a program with a specification at runtime. Some of these fall within the realm of automatic runtime consistency checking while others are beyond its scope. The following rule defines what kind of analyses and tests need to be performed:

> *Inconsistencies must be detected as soon as possible (after they occur) during the execution of the program*

The specifications impose constraints on the program at various points during its execution. If at any of these points, the program execution does not satisfy the constraints imposed at that point, then there is an *inconsistency.* The inconsistency is considered to occur only *after* the program execution reaches this point. This is an important consideration for the purpose of this thesis. There may be a confusion, for after all, if the program execution does reach a point where it becomes inconsistent with its specifications, then the program must have been inconsistent with its specifications in the first place. That is, the inconsistency existed ever since the program was written! The confusion is resolved by noting that there are two different notions of inconsistency being used here. On the one hand, the *program execution* becomes inconsistent with its specifications when (and only when) it reaches a point where the program does not satisfy the constraints imposed at that point; while on the other hand, a *program* is considered inconsistent with its specifications if it is possible for the program execution to reach a point where it does not satisfy the constraints imposed at this point. Note that program execution inconsistency on a particular test-data implies program inconsistency but not the other way around. That is, a program that is inconsistent with its specifications may perform correctly for certain test-data. This thesis deals with methods to detect program execution inconsistencies. Unless explicitly stated, the term "consistency" in this thesis is used to refer to that with respect to program execution.

As a consequence of the rule stated above, a test needs to be performed after each of the two assignments in Example 1.1 to check for the consistency of the program execution. In this simple example, there is scope for optimization for it is quite obvious that execution of

the first statement does not cause an inconsistency, while execution of the second statement does. Note however, that such optimizations, though useful, are not required of runtime consistency checking.

In Example 1.2, a test needs to be performed after the assignments to E_1, D_2 and D_7. As has been pointed out in Chapter 1, it will not be possible in the general case to determine before the program starts execution when such tests need to be performed. In the general case, messages have to be sent to a theorem prover which in turn decides when tests are necessary. The theorem prover bases its decision on its knowledge of the algebraic specification and the messages it has received so far. If a test has to be performed, the theorem prover sends the appropriate message to the checking system (see Figure 1.1). Though it is possible to determine ahead of time that tests have to be performed after the assignments to E_1, D_2 and D_7, such optimizations are not required of runtime consistency checking.

There is a second rule that characterizes the kinds of analyses and tests that do not have to be performed. This rule is stated below:

- *Inconsistencies that do not actually occur during the execution of the program need not be detected*

This rule can be rephrased as saying that apart from the tests that need to be performed as a consequence of the first rule, no other tests have to be performed. For example, it is not necessary to predict possible future inconsistencies. Hence, in Example 1.1, an analysis that shows the evaluation of the second assignment will always result in an inconsistent state need not be performed (one of the optimizations mentioned earlier). Though such extra tests are certainly useful, they are beyond the scope of runtime consistency checking and this thesis. Many other forms of program validation do attempt to perform these tests. Static program verification in fact attempts to make such predictions for all possible executions. Limiting the scope of the problem facilitates in the design of more efficient consistency checking strategies.

Chapter 3

Generalized Assertions Checking

This chapter deals with the details of automatic runtime consistency checking of programs with respect to generalized assertions. The next chapter deals with algebraic specification checking.

3.1 Overview

In Anna, object annotations, subtype annotations, statement annotations, subprogram annotations, result annotations and exception annotations are all generalizations of assertions. These constructs fall into two categories—*local constraints*, which constrain a particular point in the program, and *global constraints*, which are constraints over a larger portion of the program. In Anna, simple statement annotations, exception annotations and *out* annotations are local constraints[1]; all other annotations described in this thesis are global constraints. While tests corresponding to local constraints are expanded inline, global constraints are transformed into *checking functions*. Checking functions are described in Section 3.2.

The transformation takes place in three logical steps. The first step is to transform annotations into more basic annotations. Subtype annotations, object annotations and result annotations are considered the basic annotations. Note that *out* annotations are a form of object annotations and are therefore also basic annotations. A simple transformation

[1] Though exception annotations and out annotations do constrain *all* exit points from the scope, they are still categorized as local constraints for the purpose of this thesis, since they are somewhat more local than the other global constraints. Also, as will be seen later, the transformation methodologies of these different forms of local constraints are similar.

scheme is sufficient to reduce all the Anna constructs into the basic constructs mentioned above. These transformations are described in Section 3.3. The second step involves a few preliminary transformations. The most important of the preliminary transformations is the transformation of Anna expressions to Ada expressions. This is discussed in Section 3.4. The third step is to transform the basic annotations into checking code. This is when the checking functions are generated. This is described in Section 3.5.

Note that these three steps are interleaved. For example, the transformation of the Anna membership test **isin** requires the appropriate checking function to have already been generated; while the generation of a checking function containing an Anna membership test requires the membership test to have already been transformed.

3.2 Checking Functions

A checking function is an Ada function that checks for the validity of annotation(s) and takes an appropriate action. This action could either be to return a BOOLEAN value that specifies the result of the check, or to raise an exception if the check shows that the annotation was violated. At various places where annotations can potentially be violated, the appropriate checking functions are called. The advantages of implementing checks as calls to checking functions instead of expanding the checks inline are quite apparent. It is much simpler to generate checking functions and calls to them since the annotations do not have to be stored for too long by the Anna Transformer; only the names of the generated checking functions have to be stored. Also, this decreases the dependence on the visibility rules of the language. Since it is usually the case that the same annotation has to be checked for validity at more than one place, the checking function approach lowers space requirements. However, the effect of expanding checks inline can still be achieved by using compiler directives.

Any level in an Anna program is constrained by annotations at the current level and by annotations declared in more global levels. For example, an Ada subtype is constrained by its own subtype annotation and by the subtype annotation of its base type. In this situation, the checking functions at the more nested level make calls to checking functions at the outer levels. This ensures that when this new checking function is called, both annotations are checked as required by the Anna semantics.

Checking functions are defined for the three basic kinds of annotations. Checking functions that raise an exception (if the annotation is found to be FALSE) are generated for each of the above kinds of annotations. The structure of these checking functions is shown below; the braces indicate zero or more occurrences of the enclosed entity:

```
function CHECKING_FUNCTION(X:T) return T is
   declarations
begin
  {if not (annotation) then
      raise ANNA_EXCEPTION;
   end if;}
   return X;
exception
   when ANNA_EXCEPTION =>
      report_violation;
      raise ANNA_ERROR;
   when ANNA_ERROR =>
      raise;
   when others =>
      report_Ada_error;
      report_violation;
      raise ANNA_ERROR;
end CHECKING_FUNCTION;
```

All checking functions that raise exceptions fit into the above template. This template is for a checking functions that checks a subtype annotation of type T; an object annotation in which there is a variable of type T; or a result annotation of a function that returns a value of type T. The declarations in the checking functions are typically Ada *renaming* declarations, which though not essential, makes the generation of the rest of the checking function easier. The *if* statement evaluates the annotation (with the Anna portions transformed to Ada) and raises the exception ANNA_EXCEPTION if the annotation is FALSE. As the braces around this statement indicates, a checking function could check for the validity of more than one annotation. If all annotations evaluate to TRUE, the checking function returns the value that was passed to it, namely X, thus acting as a no-op. Typically, the formal parameter X occurs in the annotations being tested. If this checking function has to call another checking function, the return statement above is replaced by:

```
return THE_OTHER_CHECKING_FUNCTION(X);
```

Each checking function has three exception handlers. The first is a handler for the exception ANNA_EXCEPTION. ANNA_EXCEPTION is raised if this checking function detects an Anna constraint violation. In this case, a violation is reported and the exception ANNA_ERROR is raised. The second handler is for the exception ANNA_ERROR. ANNA_ERROR is raised as a result of a call to another checking function from this checking function. If this other checking function detects an inconsistency, it returns by raising ANNA_ERROR. The handler for ANNA_ERROR simply reraises this exception. The third exception handler handles all other exceptions. Typically these exceptions are raised due to Ada errors (such as division by 0) that may occur during the evaluating of the expression corresponding to the annotation. This handler reports this situation and then raises the exception ANNA_ERROR.

For subtype annotations there is one additional checking function defined. This function returns TRUE if the annotation is satisfied, and it returns FALSE otherwise. This checking function is used to implement the Anna membership test **isin**. It is also used to check the initial values of variables and formal parameters. The structure of this checking function is shown below:

```
function ISIN_CHECKING_FUNCTION(X:T) return BOOLEAN is
begin
   return annotation;
exception
   when others =>
      return FALSE;
end ISIN_CHECKING_FUNCTION;
```

This form of checking function is much simpler than the previous one. The annotation[2] is evaluated, and this value is returned. In case an exception is raised during the evaluation of the annotation, it is handled within this function and the function returns the value FALSE. In case another checking function needs to be called from this checking function, the first *return* statement above is replaced by:

[2]Note that the second form of checking function evaluates only one annotation. It will be seen later that in the case of subtype annotations, even the first form of checking functions evaluates only one annotation.

return *annotation* **and** THE_OTHER_ISIN_CHECKING_FUNCTION(X);

All checking functions are generated at the beginning of the later declarative region of the declarative region where the annotation is declared. Further details of each kind of checking function and their use are described later. The checking function that is actually generated by the Anna Transformer is slightly more complex than what has been shown here. This extra complexity provides a more comprehensive interaction with the Anna Debugger. Some of the additional features are described briefly below, but will not be discussed further in this thesis:

1. Checking functions first find out from the Anna Debugger whether or not the annotations to be checked are currently suppressed. If so, no checks are performed.

2. Checking functions have additional parameters which contain information regarding the location from where the call to them was made. This information along with the information regarding the location of the annotation is sent to the Anna Debugger in case an inconsistency is detected.

3. Checking functions allow the Anna Debugger to make decisions regarding whether or not to raise the exception ANNA_ERROR.

Figure 3.1 illustrates the transformation of basic annotations to checking functions[3].

Example 3.1: An example of the transformation of subtype annotations to checking functions is now shown. First, the Anna program before transformation is shown below:

```
procedure P is
   subtype EVEN is INTEGER;
   --| where X:EVEN => X mod 2 = 0;
   subtype POS_EVEN is EVEN;
   --| where X:POS_EVEN => X > 0;
begin
   ...
end P;
```

[3]Solid arrows with a black box (■) represents a transformation. Dashed arrows indicate that the corresponding entities are just moved over. This convention will be used in all figures in this chapter.

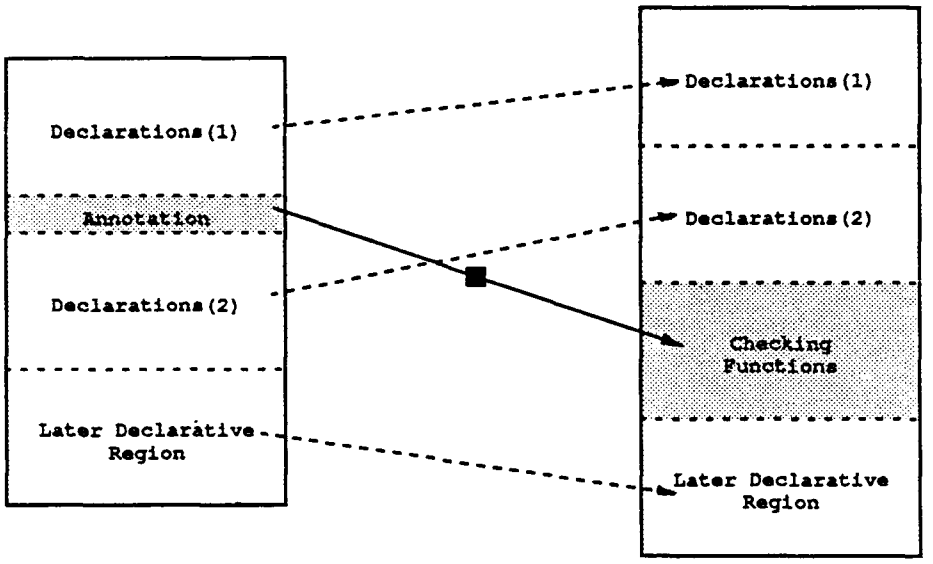


Figure 3.1: Transformation of Annotations to Checking Functions

The transformed Ada program is now shown below. Notice the two different kinds of checking functions for each of the two types, and the calls to checking functions corresponding to type EVEN from the checking functions corresponding to type POS_EVEN:

```
procedure P is
   subtype EVEN is INTEGER;
   subtype POS_EVEN is EVEN;

   function EVEN_CHECKING_FUNCTION(X:T) return T is
   begin
      if not (X mod 2 = 0) then
         raise ANNA_EXCEPTION;
      end if;
      return X;
   exception
      when ANNA_EXCEPTION =>
         report_violation;
         raise ANNA_ERROR;
      when ANNA_ERROR =>
         raise;
```

```
    when others =>
        report_Ada_error;
        report_violation;
        raise ANNA_ERROR;
end EVEN_CHECKING_FUNCTION;

function EVEN_ISIN_CHECKING_FUNCTION(X:T) return BOOLEAN is
begin
    return X mod 2 = 0;
exception
    when others =>
        return FALSE;
end EVEN_ISIN_CHECKING_FUNCTION;

function POS_EVEN_CHECKING_FUNCTION(X:T) return T is
begin
    if not (X > 0) then
        raise ANNA_EXCEPTION;
    end if;
    return EVEN_CHECKING_FUNCTION(X);
exception
    when ANNA_EXCEPTION =>
        report_violation;
        raise ANNA_ERROR;
    when ANNA_ERROR =>
        raise;
    when others =>
        report_Ada_error;
        report_violation;
        raise ANNA_ERROR;
end POS_EVEN_CHECKING_FUNCTION;

function POS_EVEN_ISIN_CHECKING_FUNCTION(X:T) return BOOLEAN is
begin
    return X mod 2 = 0 and EVEN_ISIN_CHECKING_FUNCTION(X);
```

```
    exception
        when others =>
            return FALSE;
    end POS_EVEN_ISIN_CHECKING_FUNCTION;

begin
    ...
end P;
```

3.3 Transformation to Basic Annotations

3.3.1 Simple Statement Annotations

A simple statement annotation is an *out* annotation on the immediately preceding statement. If the annotation occurs before the first statement in a sequence of statements, annotation is a constraint on an imaginary *null* statement just before the annotation.

Simple statement annotations are transformed to *out* annotations. A *block* statement is created at the location of the annotation. The annotation is converted to an *out* annotation and placed in the declarative region of this *block* statement. The statement constrained by the annotation is placed in the statement part of the *block* statement. Figure 3.2 illustrates this transformation.

Example 3.2:

Before:

```
procedure P is
    A:INTEGER := 3;
begin
    --| A = 3;
    A := A + 1;
    --| A = in A + 1;
end P;
```

After:

```
procedure P is
    A:INTEGER := 3;
begin
```

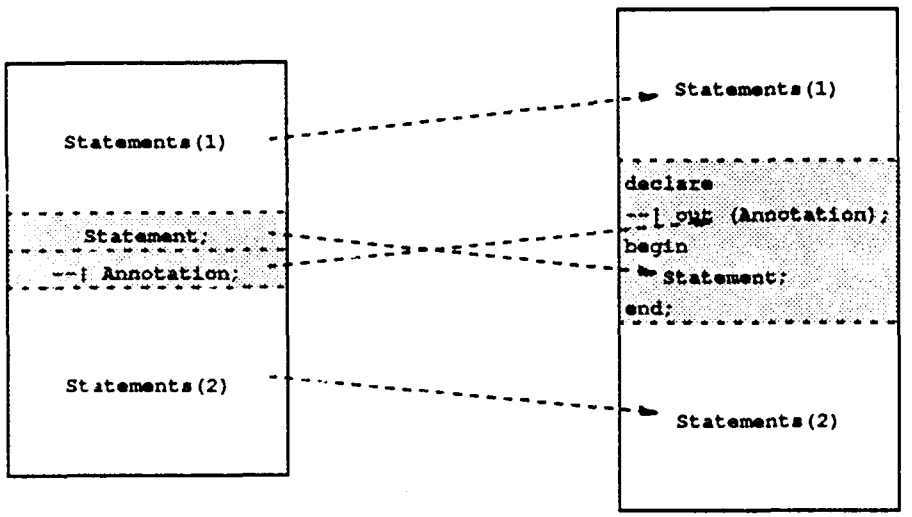


Figure 3.2: Transformation of Simple Statement Annotations

```
declare
   --| out (A = 3);
begin
   null;
end;
declare
   --| out (A = in A + 1);
begin
   A := A + 1;
end;
end P;
```

Simple statement annotations are not expanded directly into checks at the location of the annotation. This is because of the possibility that control never reaches this point, and therefore the check is never made. This can happen when the previous statement transfers control to some other point. Examples of such statements are *return* statements, *raise* statements, *exit* statements and *goto* statements.

3.3.2 Compound Statement Annotations

Compound statement annotations have a compound statement immediately following them, and this statement is the scope of the annotation. If the compound statement is not a block statement, a new block statement is inserted in the place of the original compound statement. The original compound statement is now placed in the statement part of this new block statement. The compound statement annotation is transformed to an object annotation and placed in the declarative region of the block statement. If the compound statement is already a block statement, then the compound statement annotation is transformed to an object annotation and placed at the beginning of the declarative region of this block statement. No new block statement is created in this case. Figure 3.3 illustrates this transformation.

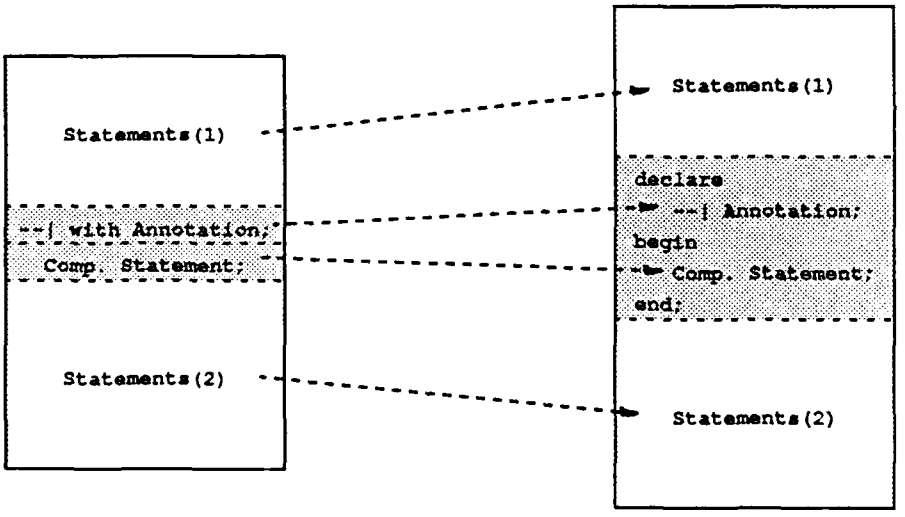


Figure 3.3: Transformation of Compound Statement Annotations

Example 3.3:

Before:

```
--| with
--|      A mod 2 = 0;
for I in J..K loop
```

```
   A := A+2;
end loop;
```

After:

```
declare
   --| A mod 2 = 0;
begin
   for I in J..K loop
      A := A+2;
   end loop;
end;
```

3.3.3 Subprogram Annotations

Subprogram annotations are transformed to object annotations and placed at the beginning of the declarative region of the subprogram. Figure 3.4 illustrates this transformation.

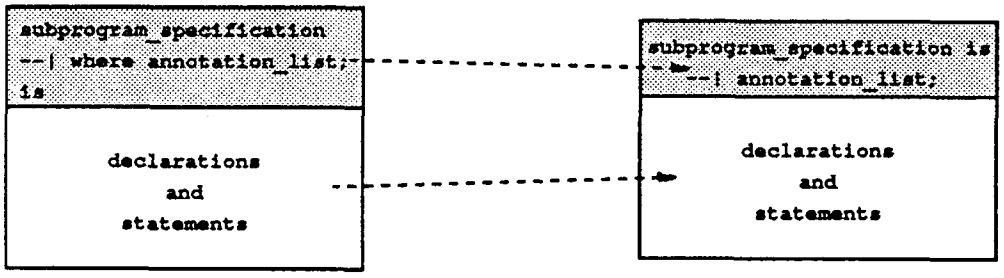


Figure 3.4: Transformation of Subprogram Annotations

Example 3.4:

Before:

```
procedure P(X:in out EVEN)
--| where
--|     X mod 2 = 0;
is
```

```
begin
   X := X+2;
end P;
```

After:

```
procedure P(X:in out EVEN) is
   --| X mod 2 = 0;
begin
   X := X+2;
end P;
```

3.4 Transformation of Anna Expressions

This section describes in detail the transformation of Anna expressions to Ada expressions. Some of the transformations results in the introduction of new variables and aliases to already existing variables. Hence these transformations generate declarations. The transformations described in Section 3.3 guarantee that all annotations after these transformations will occur in one of the basic declarative regions of the program. Hence the declarations can be inserted just prior to the annotation whose expression is being transformed. The transformations permit the transformed expression to be moved anywhere in the region defined by the scope of the original annotation. Figure 3.5 illustrates this transformation process.

Anna expressions are transformed to Ada expressions through a five stage process. Each of these stages is discussed in separate sections below.

3.4.1 Stage 1: Implication Operators and Anna Membership Operators

The Anna implication operators –> and <–> are equivalent to the predefined Ada relational operators (whose arguments are of type BOOLEAN) <= and = respectively. However, the Anna implication operators have lower precedence than the Ada relational operators. The precedence information is available in the parse tree and therefore poses no problems for the Anna Transformer. The Anna implication operators are transformed to function calls to the corresponding predefined Ada relational operators as illustrated in the following examples:

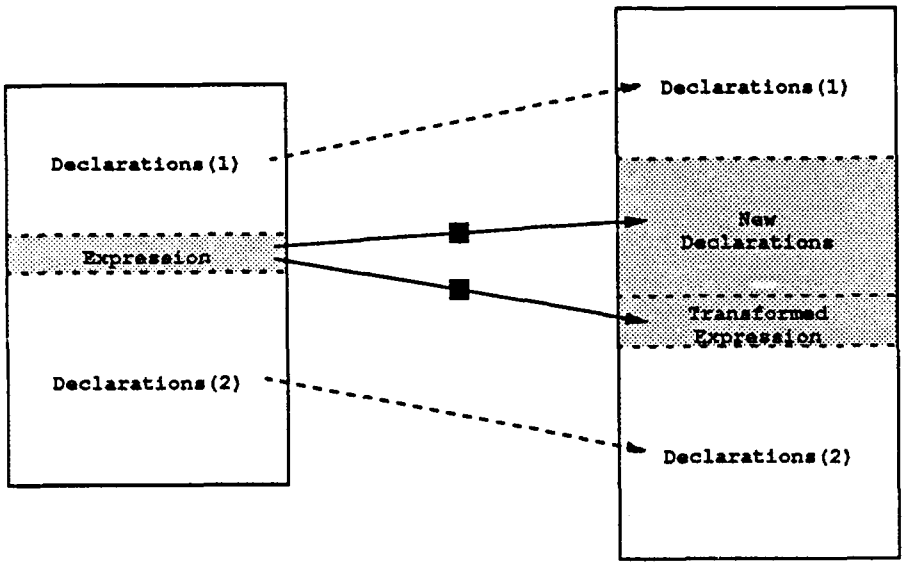


Figure 3.5: Transformation of Anna Expressions

Example 3.5:

	Before:	*After*:
i.	A < B −> B > A	STANDARD."<="(A < B, B > A)
ii.	EVEN(X) <−> **not** ODD(X)	STANDARD."="(EVEN(X), **not** ODD(X))

The Anna membership operator **isin** is equivalent to a combination of the Ada membership operator **in** and possibly a call to a checking function. The call to a checking function is generated only when a type name is used in the membership operator and a checking function exists for this type name.

Example 3.6:

	Before:	*After*:
i.	X **isin** INTEGER	X **in** INTEGER
ii.	X **isin** EVEN	X **in** EVEN **and** EVEN_ISIN_CHECKING_FUNCTION(X)
iii.	X **isin** MIN..MAX	X **in** MIN..MAX

In the above example, MIN and MAX are assumed to be of type INTEGER. Type EVEN is assumed to be defined as in Example 3.1.

3.4.2 Stage 2: Conditional Expressions

A conditional expression in Anna has the following syntax:

```
if condition then
    expression
{elsif condition then
    expression}
else
    expression
end if
```

Hence, if a conditional expression has n conditions, then it has $n+1$ expressions. For the purpose of this section, a simplified notation is used to denote conditional expressions. Let $\mathbf{C} = \langle C_1, \ldots, C_n \rangle$ be the list of all the conditions and $\mathbf{E} = \langle E_1, \ldots, E_{n+1} \rangle$ be the list of all the expressions of the conditional expression. Then the conditional expression is denoted by $\mathcal{C}(\mathbf{C}, \mathbf{E})$.

The two expressions $f(\mathcal{C}(\mathbf{C}, \mathbf{E}))$ and $\mathcal{C}(\mathbf{C}, f(\mathbf{E}))$ are equivalent except in three specific cases which will be described later. Here, $f(\mathbf{E})$ denotes $\langle f(E_1), \ldots, f(E_{n+1}) \rangle$. This is true because the evaluation of f cannot have any side-effects since it is part of an annotation[4]. This equivalence is used to repeatedly transform the conditional expression until its type (the type of the E_i's) is BOOLEAN. This is guaranteed to happen because all annotations are of type BOOLEAN. Hence, when the conditional expression is moved out all the way, its type becomes BOOLEAN. Transforming BOOLEAN conditional expressions to Ada expressions will be discussed later. An example of this transformation is shown below. First a set of type and variable declarations is given below which is used in the examples of this section:

```
type RELIGION_TYPE is (ISLAM, JUDAISM, CHRISTIANITY);
type SABBATH_DAYS is (FRIDAY, SATURDAY, SUNDAY, NONE);
type RELIGION_REC(ATHIEST: BOOLEAN) is record
   case ATHIEST is
      when TRUE =>
         null;
```

[4] An Anna requirement.

```
        when FALSE =>
            RELIGION:RELIGION_TYPE;
            SABBATH:SABBATH_DAYS;
    end case;
end record;

R:RELIGION_REC;
S:SABBATH_DAYS;
```

Example 3.7: This example illustrates the transformation of conditional expressions to BOOLEAN conditional expressions. In this example, $f(x) \equiv (\mathrm{R.SABBATH} = x)$:

Before:

```
R.SABBATH = if R.RELIGION = ISLAM then
                FRIDAY
            elsif R.RELIGION = JUDAISM then
                SATURDAY
            else
                SUNDAY
            end if;
```

After:

```
if R.RELIGION = ISLAM then
    R.SABBATH = FRIDAY
elsif R.RELIGION = JUDAISM then
    R.SABBATH = SATURDAY
else
    R.SABBATH = SUNDAY
end if;
```

The three cases in which the equivalence $f(\mathcal{C}(\mathbf{C},\mathbf{E})) = \mathcal{C}(\mathbf{C}, f(\mathbf{E}))$ does not hold are now discussed. In these cases, the above-mentioned transformation cannot be performed. Variations of this transformation are therefore provided for each of the cases.

Case 1: f encloses its argument within an *initial expression*. That is, $f(x)$ is of the form $g(\mathbf{in}(h(x)))$. In this case, the expressions $f(\mathcal{C}(\mathbf{C},\mathbf{E}))$ and $\mathcal{C}(\mathbf{in}(\mathbf{C}), f(\mathbf{E}))$ are equivalent, and therefore a similar transformation can still be performed.

Example 3.8: This example is quite similar to Example 3.7, except that the expression is an initial expression.

Before:

```
in (R.SABBATH = if R.RELIGION = ISLAM then
                    FRIDAY
                elsif R.RELIGION = JUDAISM then
                    SATURDAY
                else
                    SUNDAY
                end if
   );
```

After:

```
if in (R.RELIGION = ISLAM) then
   in (R.SABBATH = FRIDAY)
elsif in (R.RELIGION = JUDAISM) then
   in (R.SABBATH = SATURDAY)
else
   in (R.SABBATH = SUNDAY)
end if;
```

Case 2: f contains an Ada short-circuit operation. For example, $f(x)$ could be of the form (g **and then** $h(x)$). In this case, x should not be evaluated if g happens to be FALSE. The normal transformation would result in $\mathcal{C}(\mathbf{C}, g$ **and then** $h(\mathbf{E}))$ which is incorrect, for **C** is evaluated even if g is FALSE. However, this is not a problem since the arguments of an Ada short-circuit operation are of type BOOLEAN. Therefore, a BOOLEAN conditional expression can be obtained in this case without having to move the Ada short-circuit operation into the conditional expression.

Example 3.9: In this example, the second expression, which has been derived from the first using the normal transformations, is not equivalent to the first. However, the third expression, in which the conditional expression is BOOLEAN, is equivalent to the first. In this case, the short-circuit operation has not been moved into the conditional expression.

```
1.  not R.ATHIEST and then R.SABBATH = if R.RELIGION = ISLAM then
                                           FRIDAY
                                       elsif R.RELIGION = JUDAISM then
                                           SATURDAY
                                       else
                                           SUNDAY
                                       end if;
```

```
2.  if R.RELIGION = ISLAM then        -- Wrong!
        not R.ATHIEST and then R.SABBATH = FRIDAY
    elsif R.RELIGION = JUDAISM then
        not R.ATHIEST and then R.SABBATH = SATURDAY
    else
        not R.ATHIEST and then R.SABBATH = SUNDAY
    end if;
```

```
3.  not R.ATHIEST and then if R.RELIGION = ISLAM then
                               R.SABBATH = FRIDAY
                           elsif R.RELIGION = JUDAISM then
                               R.SABBATH = SATURDAY
                           else
                               R.SABBATH = SUNDAY
                           end if;
```

Case 3: f encloses its argument within a conditional expression. If the argument to f is also a conditional expression, then there is a nesting of conditional expressions. Moving the inner conditional expression outside changes the meaning of the whole expression and so the normal transformations cannot be performed. There are two ways in which the conditional expressions can be nested: $f(x)$ is such that x is within the condition part of a conditional expression. For example, $f(x)$ could be (**if** $g(x)$ **then** ... **end if**). In this case, since $g(x)$ has to be BOOLEAN a BOOLEAN conditional expression can be obtained without having to move the conditional expression x outside the enclosing conditional expression. The second possibility is that x is within the expression part of a conditional expression. For

example, $f(x)$ could be of the form (**if** g **then** $h(x)$ ... **end if**). In this case, the outer conditional expression is transformed completely first, and when this is done, $h(x)$ will have been transformed to a BOOLEAN expression.

Example 3.10: In this example (as in the case of Example 3.9, the second expression is not equivalent to the first. However, the third expression in which all conditional expressions are BOOLEAN is equivalent to the first. Note that only the inner conditional expression has been transformed in the second expression.

1.
```
S = if not R.ATHIEST then
        if R.RELIGION = ISLAM then
            FRIDAY
        elsif R.RELIGION = JUDAISM then
            SATURDAY
        else
            SUNDAY
        end if
    else
        NONE
    end if;
```

2.
```
if R.RELIGION = ISLAM then          -- Wrong!
    S = if not R.ATHIEST then
            FRIDAY
        else
            NONE
        end if
elsif R.RELIGION = JUDAISM then
    S = if not R.ATHIEST then
            SATURDAY
        else
            NONE
        end if
else
    S = if not R.ATHIEST then
            SUNDAY
        else
```

```
            NONE
         end if
      end if;
```

```
3.  if not R.ATHIEST then
       if R.RELIGION = ISLAM then
          S = FRIDAY
       elsif R.RELIGION = JUDAISM then
          S = SATURDAY
       else
          S = SUNDAY
       end if
    else
       S = NONE
    end if;
```

The rest of this section discusses how BOOLEAN conditional expressions are transformed into Ada expressions. In this transformation, Ada short-circuit operations are used extensively. This transformation causes some expressions to be evaluated more than once, but this is an acceptable overhead in most situations.

Let $\mathcal{N}(i,j)$ denote (**not** C_i **and then not** C_{i+1} ... **and then not** C_j). The BOOLEAN conditional expression $\mathcal{C}(\mathbf{C},\mathbf{E})$ is transformed to:

$$
\begin{array}{l}
(\\
\quad (\qquad\qquad\qquad\quad C_1 \ \textbf{and then}\ E_1\ ) \\
\textbf{or else}\ (\mathcal{N}(1,1) \ \textbf{and then}\ C_2 \ \textbf{and then}\ E_2\ ) \\
\ldots \\
\textbf{or else}\ (\mathcal{N}(1,i-1) \ \textbf{and then}\ C_i \ \textbf{and then}\ E_i\ ) \\
\ldots \\
\textbf{or else}\ (\mathcal{N}(1,n-1) \ \textbf{and then}\ C_n \ \textbf{and then}\ E_n\ ) \\
\textbf{or else}\ (\mathcal{N}(1,n) \qquad\qquad\quad \textbf{and then}\ E_{n+1}) \\
)
\end{array}
$$

Example 3.11: This example illustrates the transformation of a BOOLEAN conditional expression to an Ada expression.

Before:

```
if R.RELIGION = ISLAM then
   R.SABBATH = FRIDAY
elsif R.RELIGION = JUDAISM then
   R.SABBATH = SATURDAY
else
   R.SABBATH = SUNDAY
end if;
```

After:

```
(
        (R.RELIGION = ISLAM
                and then R.SABBATH = FRIDAY)
 or else (not R.RELIGION = ISLAM and then R.RELIGION = JUDAISM
                and then R.SABBATH = SATURDAY)
 or else (not R.RELIGION = ISLAM and then not R.RELIGION = JUDAISM
                and then R.SABBATH = SUNDAY)
)
```

3.4.3 Stage 3: Anna Relational Expressions

Anna relational expressions have the following syntax:

expression { *relational_operator expression* }

The Anna relational expression is transformed into a conjunction of Ada relational operators.

Example 3.12:

Before:

```
I <= J < K < N
```

After:

```
STANDARD."and"(STANDARD."and"(I<=J,J<K),K<N)
```

3.4.4 Stage 4: Initial Names and Expressions

Initial names and expressions are transformed by introducing a constant declaration just before the annotation within which the initial name/expression occurs, and replacing the initial name/initial expression in the annotation by that constant. The constant is initialized to the value of the initial name or expression.

Example 3.13: Assume all variables to be of type INTEGER.

1. *Before*:

```
--| A = in A;
```

After:

```
NEW_A:constant INTEGER := A;
--| A = NEW_A;
```

2. *Before*:

```
--| out (C*C = in (A*A + in (B*B)));
```

After:

```
NEW_EXP:constant INTEGER := A*A + B*B;
--| out (C*C = NEW_EXP);
```

3.4.5 Stage 5: Renaming of Objects

To prevent confusion when expressions are moved around during the transformation process, every object that occurs in an annotation is renamed by a unique identifier and this identifier is used to refer to the object in the annotation.

Example 3.14: Assume X is of type INTEGER.

Before:

```
--| X mod 2 = 0;
```

After:

```
NEW_X:INTEGER renames X;
function NEW_MOD(X,Y:INTEGER) return INTEGER renames "mod";
function NEW_EQUAL(X,Y:INTEGER) return INTEGER renames "=";
--| NEW_EQUAL(NEW_MOD(NEW_X,2),0);
```

3.4.6 Quantified and State Expressions

Anna quantified expressions and Anna state expressions are not handled completely by the current version of the Anna Transformer. Future versions are expected to implement transformations for these expressions. In the case of quantified expressions, the programmer may have to provide *iterators* for the quantifier domains. The quantified expressions can then be transformed to use these iterators. For example, in the following quantified expression,

```
for all S:SYMBOL => MEMBER(S,SYMBOL_TABLE) -> DECLARED(S);
```

the programmer may be asked to provide an iterator over the SYMBOL_TABLE that returns all SYMBOL's that are MEMBER's of the SYMBOL_TABLE. The general solution to implementing runtime checking for quantified expressions is however a topic for future research.

State expressions do not pose the same kinds of problems as quantified expressions. They can be implemented by generating functions that take as parameters the state descriptors and return the value of the state expression. A pilot implementation for array states has been completed.

Quantified expressions and state expressions that are part of Anna package axioms are handled in a different manner. This is discussed in Chapter 4.

3.5 Generation of Checking Code

3.5.1 Checking Functions of Subtype Annotations

Every type[5] has two checking functions associated with it, whether or not they have a corresponding subtype annotation. Example 3.1 in page 22 illustrates these checking functions

[5]Here the word "type" refers to both Ada types and subtypes

for the case when the type had a corresponding subtype annotation. In the case of type without a subtype annotation, the purpose of a checking function is to make a call to a parent checking function. The reason to introduce this level of indirection is due to the visibility rules of Ada. The parent checking functions may not be visible at all locations from where it needs to be called. Hence the checking functions of a type without a corresponding subtype annotation can be thought of as bridging the gap between other checking functions and the locations from where they are called. If there is no parent checking function and the type does not have a corresponding subtype annotation, checking functions do not have to be generated. All variables that occur in subtype annotations are replaced by their values upon elaboration of the annotation. This is done by converting all variables to initial names and then performing the transformation described in Section 3.4.4. The features of checking functions of subtype annotations are now summarized:

1. There are two different kinds of checking functions.
2. There are no declarations.
3. The values of all variables are "frozen" at elaboration time.
4. There is either no annotation checked, in which case there is a call to a parent checking function; or there is exactly one annotation checked (since a type can have only one subtype annotation), in which case there may or may not be a call to a parent checking function.
5. The exception handlers are as shown in Section 3.2.

3.5.2 Checking Functions of Object Annotations

Checking functions of object annotations are the most complex of the checking functions. Only the checking functions that raise exceptions are generated. These checking functions are associated with the variables that occur in the annotations. For example, if an object annotation has three variables, this results in the generation of three checking functions, one for each of these variables. However, in any particular declarative region, there is at most one checking function per variable. This checking function checks all object annotations in this declarative region in which its corresponding variable occurs. If the variable has been declared in a more global declarative region, then there may be other object annotations constraining this variable at more global declarative regions. Hence, there can be different

checking functions for the same variable at different declarative regions. Each checking function of a variable includes a call to the next most global checking function of the same variable (if such a checking function exists). As a result of the language rules, there are no problems due to visibility rules, unlike in the case of subtype annotations. This is also the case with result annotations (see Section 3.5.3).

Example 3.15: This example illustrates the generation of checking functions for object annotations. Note the use of renaming declarations within the checking functions. This simplifies the generation of the checking functions.

Before:

```
procedure P is
   A,B:INTEGER;
   --| A+B > 0;
   --| B > 0;
   procedure Q is
      C:INTEGER;
      --| C > B;
   begin
      ...
   end Q;
begin
   ...
end P;
```

After:

```
procedure P is
   A,B:INTEGER;

   -- The following renaming declarations are a result of ap-
   -- plying the transformations of Section 3.4.5 when applied
   -- to the expressions of the first two annotations in the ex-
   -- ample.
   A1:INTEGER renames A;
   B1:INTEGER renames B;
   B2:INTEGER renames B;
```

```
-- The renaming declarations within the checking functions
-- hide the appropriate global renaming declarations.
function A_CHECKING_FUNCTION(X:INTEGER) return INTEGER is
   A1:INTEGER renames X;
begin
   if not (A1 + B1 > 0) then
      raise ANNA_EXCEPTION;
   end if;
   return X;
exception
   ...
end A_CHECKING_FUNCTION;

function B_CHECKING_FUNCTION1(X:INTEGER) return INTEGER is
   B1:INTEGER renames X;
   B2:INTEGER renames X;
begin
   if not (A1 + B1 > 0) then
      raise ANNA_EXCEPTION;
   end if;
   if not (B2 > 0) then
      raise ANNA_EXCEPTION;
   end if;
   return X;
exception
   ...
end B_CHECKING_FUNCTION1;

procedure Q is
   C:INTEGER;
   C1:INTEGER renames C;
   B3:INTEGER renames B;

   function C_CHECKING_FUNCTION(X:INTEGER) return INTEGER is
      C1:INTEGER renames X;
   begin
      if not (C1 > B3) then
         raise ANNA_EXCEPTION;
      end if;
```

```
        return X;
    exception
        ...
    end C_CHECKING_FUNCTION;

    function B_CHECKING_FUNCTION2(X:INTEGER) return INTEGER is
        B3:INTEGER renames X;
    begin
        if not (C1 > B3) then
            raise ANNA_EXCEPTION;
        end if;
        return B_CHECKING_FUNCTION1(X);
    exception
        ...
    end B_CHECKING_FUNCTION2;

begin
    ...
end Q;

begin
    ...
end P;
```

In the case of composite objects, *checking procedures* are generated instead of checking functions. This improves the performance of the checking code. Note that checking procedures are not generated for subtype annotations and result annotations even though there may be an improvement in the performance of the checking code. In the case of composite objects, checking functions can be completely replaced by checking procedures, but in the case of subtype annotations and result annotations, checking functions will still be necessary even if checking procedures were generated. A design decision was made to generate only one kind of checking subprogram for each case to avoid a proliferation of checking subprograms. In any case, the performance of the checking functions can be improved by using a compiler directive to expand these functions inline.

Example 3.16:

Before:

```
procedure P is
   type T is array(1..10) of INTEGER;
   X:T;
   --| SUM(X,1,5) > SUM(X,6,10);
begin
   ...
end P;
```

After:

```
procedure P is
   type T is array(1..10) of INTEGER;
   X:T;
   procedure X_CHECKING_PROCEDURE(X:in out T) is
   begin
      if not (SUM(X,1,5) > SUM(X,6,10) then
         raise ANNA_EXCEPTION;
      end if;
      return;
   end X_CHECKING_PROCEDURE;
begin
   ...
end P;
```

The features of checking functions of object annotations are now summarized:

1. Only the checking function that raises exceptions is generated.

2. Renaming declarations are generated within the checking function to simplify the generation of the rest of the checking function.

3. These checking functions correspond to variables, and are generated in any declarative region where there is an object annotation constraining this variable.

4. Within a declarative region, there is only one checking function for any particular variable which checks all object annotations within the same declarative region that constrain this variable.

5. The exception handlers are as shown in Section 3.2.

6. In the case of composite objects, checking procedures are generated instead of checking functions.

3.5.3 Checking Functions of Result Annotations

Result annotations can occur as subprogram annotations, object annotations or statement annotations. However, as a result of the transformations described in Section 3.3, all result annotations are guaranteed to be object annotations, i.e., they will occur only within a declarative region. Checking functions of result annotations are similar to those of object annotations. There is a renaming declaration within the checking function corresponding to each annotation checked by the checking function. Also, in every declarative region that contains result annotations there is one result annotation that checks these annotations. Unlike in the case of object annotations where object annotations can constrain more than one variable, result annotations can constrain only one function—the most local function that encloses the annotation. Also, all result annotations within the same declarative region constrain the same function. Hence at most one checking function requires to be generated in each declarative region. Just as in the case of object annotations, result annotations of the same function can occur in different declarative regions. Here again, each checking function calls the next most global checking function (if one exists).

Example 3.17: This example illustrates the generation of checking functions for result annotations:

Before:

```
function LOG(F:FLOAT) return FLOAT is
   --| return A:FLOAT => A < F;
   --| return B:FLOAT => F<1.0 -> B<0.0;
begin
   ...
end LOG;
```

```
After:
function LOG(F:FLOAT) return FLOAT is

   -- The global renaming declarations are ignored for simplic-
   -- ity.

   function LOG_CHECKING_FUNCTION(X:FLOAT) return FLOAT is
      A:FLOAT renames X;
      B:FLOAT renames X;
   begin
      if not (A < F) then
         raise ANNA_EXCEPTION;
      end if;
      if not (STANDARD."<="(F < 1.0, B < 0.0)) then
         raise ANNA_EXCEPTION;
      end if;
      return X;
   exception
      ...
   end LOG_CHECKING_FUNCTION;

begin
   ...
end LOG;
```

3.5.4 Calls to Checking Functions

In addition to the call made to a checking function to implement the Anna membership test (Section 3.4.1), checking functions are also called at places where the variables present in the corresponding annotations change value. The Ada/Anna language rules guarantee that annotations can change their values only at such places[6]. Details of all calls to checking functions are discussed below. In this discussion, all variables are assumed to be defined (i.e., they are initialized in their declarations). If this assumption is not made, it will be necessary to create a BOOLEAN variable corresponding to every program variable. These variables are

[6]This applies to sequential programs only. When there are multiple threads of control in the program, it is assumed that the execution of one thread of control does not affect the value of annotations constraining other threads of control.

all initialized to FALSE and changed to TRUE whenever the corresponding program variable is given a value. It is necessary to have such a BOOLEAN variable even for components of composite types. However, experiments have demonstrated that this assumption can be made without significantly affecting the usefulness of the Anna Consistency Checking System.

Object Declarations

A check is performed at an object declaration if the type of the object has a corresponding checking function (i.e., the type is constrained directly or indirectly by subtype annotations). This is performed by generating a dummy object declaration immediately after the object declaration in question. This object is initialized by calling a function in the Anna runtime library. Calling this function has the side-effect of performing the necessary check. This function usually calls the checking function of the type of the declared object. However, there is an Ada rule that restricts calls to subprograms until they are elaborated. Hence, if the type has been declared in the same declarative region as the object declaration, then the checking function cannot be called. In this situation, the annotation is explicitly checked. In general, there will be explicit checks corresponding to the relevant types in the same declarative region as the object declaration, and one call to a checking function declared in a more global declarative region.

Example 3.18: In this example (as in the case of future examples), the function CHECK (there is also a procedure with the same name that will occur in later examples) is defined in the Anna runtime library.

```
Before:
declare
   subtype PERFECT_CUBE is INTEGER;
   --| where X:PERFECT_CUBE =>
   --|     (INTEGER(CUBE_ROOT(FLOAT(X)))**3) = X;
begin
   declare
      subtype EVEN_PERFECT_CUBE is PERFECT_CUBE;
      --| where X:EVEN_PERFECT_CUBE =>
      --|     X mod 2 = 0;
```

```
        subtype POS_EVEN_PERFECT_CUBE is EVEN_PERFECT_CUBE;
        --| where X:POS_EVEN_PERFECT_CUBE =>
        --|     X > 0;
        A:PERFECT_CUBE := 0;
        B:EVEN_PERFECT_CUBE := 0;
        C:POS_EVEN_PERFECT_CUBE := 0;
    begin
        ...
    end;
end;
```

After:

```
declare
    subtype PERFECT_CUBE is INTEGER;
    ...
begin
    declare
        subtype EVEN_PERFECT_CUBE is PERFECT_CUBE;
        subtype POS_EVEN_PERFECT_CUBE is EVEN_PERFECT_CUBE;

        A:PERFECT_CUBE := 0;
        A_CHECK:BOOLEAN
            := CHECK(ISIN_PERFECT_CUBE_CHECKING_FUNCTION(A));

        B:EVEN_PERFECT_CUBE := 0;
        B1_CHECK:BOOLEAN := CHECK(B mod 2 = 0);
        B2_CHECK:BOOLEAN
            := CHECK(ISIN_PERFECT_CUBE_CHECKING_FUNCTION(B));

        C:POS_EVEN_PERFECT_CUBE := 0;
        C1_CHECK:BOOLEAN := CHECK(C > 0);
        C2_CHECK:BOOLEAN := CHECK(C mod 2 = 0);
        C3_CHECK:BOOLEAN
            := CHECK(ISIN_PERFECT_CUBE_CHECKING_FUNCTION(C));

        ...

    begin
```

```
        ...
    end;
end;
```

Assignment Statements

The expression on the right-hand side must satisfy any annotations on the subtype of the target variable. It also has to satisfy all object annotations on the target variable. This is checked by transforming the right-hand side expression to include calls to the appropriate checking functions.

Example 3.19:

Before:

```
declare
    A, B : EVEN;
    --| B <= 10;
begin
    A := 6;
    B := A;
end;
```

After:

```
declare
    A, B : EVEN;
    ...
begin
    A := EVEN_CHECKING_FUNCTION(6);
    B := B_CHECKING_FUNCTION(EVEN_CHECKING_FUNCTION(A));
end;
```

Entry into Subprograms

On entry into subprograms, all parameter values of mode *in* and *in out* have to satisfy any constraints on their types. The method of checking these constraints is quite similar to that of checks performed at object declarations. However, in this case, all types are declared at some global declarative region and hence no explicit checks need to be generated.

Example 3.20:

Before:

```
procedure ABSOLUTE_VALUE(X:in EVEN;Y:in out POS_EVEN) is
begin
   ...
end ABSOLUTE_VALUE;
```

After:

```
procedure ABSOLUTE_VALUE(X:in EVEN;Y:in out POS_EVEN) is
   X_CHECK:EVEN := CHECK(ISIN_EVEN_CHECKING_FUNCTION(X));
   Y_CHECK:POS_EVEN := CHECK(ISIN_POS_EVEN_CHECKING_FUNCTION(Y));
begin
   ...
end ABSOLUTE_VALUE;
```

Exit from Procedures

On exit from procedures, the values of all formal parameters of mode *in out* and *out* are assigned to the corresponding actual parameters. The new values of the actual parameters must satisfy any constraints on their types, and also constraints imposed by object annotations. Dummy assignment statements are generated immediately after the procedure call, which are then transformed like any other assignment statement. Note that since functions can have only *in* parameters, these checks do not have to be performed on exit from functions.

Example 3.21:

Before:

```
declare
   subtype SMALL_POS_EVEN is POS_EVEN;
   --| where X:SMALL_POS_EVEN =>
   --|     X <= 100;
   A:EVEN;
   B:SMALL_POS_EVEN;
   --| B > 10;
begin
```

```
    A := ...;
    ABSOLUTE_VALUE(A,B);
end;
```

After:

```
declare
    subtype SMALL_POS_EVEN is POS_EVEN;
    A:EVEN;
    B:SMALL_POS_EVEN;
    ...
begin
    A := ...;
    ABSOLUTE_VALUE(A,B);
    B := B_CHECKING_FUNCTION(SMALL_POS_EVEN_CHECKING_FUNCTION(B));
end;
```

Type Conversions and Qualified Expressions

The values of type conversions and qualified expressions have to satisfy any constraints on their corresponding types. This is checked by calling the appropriate checking function as shown in the example below.

Example 3.22:

Before:

```
declare
    A:INTEGER;
    B:EVEN;
begin
    B := EVEN(A) + EVEN'(2);
end;
```

After:

```
declare
    A:INTEGER;
    B:EVEN;
begin
```

```
    B := EVEN_CHECKING_FUNCTION(EVEN(A))
            + EVEN_CHECKING_FUNCTION(EVEN'(2));
end;
```

Return Statements

When returning from a function, the return value must satisfy any result annotation on this function. This is checked by calling the appropriate checking function.

Example 3.23:

Before:

```
function LOG(F:FLOAT) return FLOAT is
   --| return A:FLOAT => A < F;
   --| return B:FLOAT => F<1.0 -> B<0.0;
begin
   return LN(F)/LN(10.0);
end LOG;
```

After:

```
function LOG(F:FLOAT) return FLOAT is
   ...
begin
   return LOG_CHECKING_FUNCTION(LN(F)/LN(10.0));
end LOG;
```

Location of Object Annotations

At the location of all object annotations, a check has to be made to ensure that the object annotation holds at this point (the initial state of the scope of the object annotation). To perform this check, no checking function is called, rather the check is explicitly performed.

Example 3.24:

Before:

```
declare
    A,B.INTEGER := 0;
    --| A+B >= 0;
begin
    ...
end;
```

After:

```
declare
    A,B:INTEGER := 0;
    OBJ_ANNO_CHECK:BOOLEAN := CHECK(A+B >= 0);
    ...
begin
    ...
end;
```

Checking of Composite Objects

In the case of composite objects, checking functions might introduce an overhead due to the possibility of making a copy of the composite object. This overhead shows itself especially in the case of the dummy assignment statements that are used to check constraints after returning from procedures. An optimization can be performed in the case of object annotations to solve this problem. For composite objects, checking procedures are generated instead of checking functions (see Section 3.5.2). The composite object is passed to the checking procedure as an *in out* parameter. The checking procedures are called at slightly different locations as compared to the checking functions. For example, in the case of assignment statements, the checking procedure is called immediately after the assignment statement. This results in a slightly delayed, but more efficient performance of the check.

Composite objects also present another problem. When an assignment is made to a component of a composite object, this could potentially violate constraints on both the component type and the composite type in addition to any object annotations on the composite object. In this case, all relevant checks are performed as shown in the example below.

Example 3.25:

Before:

```
declare
   subtype T is INTEGER;
   --| where X:T => X > 0;
   type R is record
      C:T;
   end record;
   --| where X:R => X.C < 10;
   type A is array(1..2) of R;
   --| where X:A => X(1).C > X(2).C;
   Z:A;
   --| Z(1).C + Z(2).C > 5;
begin
   Z(1).C := exp;
end;
```

After:

```
declare
   subtype T is INTEGER;
   type R is record
      C:T;
   end record;
   type A is array(1..2) of R;
   Z:A;
   ...
begin
   Z(1).C := T_CHECKING_FUNCTION(exp);
   Z(1) := R_CHECKING_FUNCTION(Z(1));
   Z_CHECKING_PROCEDURE(A_CHECKING_FUNCTION(Z));
end
```

Something similar can be done in the case of access types also. However, in this case there is the additional problem of (dynamic) aliasing[7]. The only way in which this problem can be solved is by brute force. Every time an access object is changed, all other access objects are also checked for consistency. Since this is extremely inefficient, some other alternative has

[7]Static aliasing that is achieved by Ada renaming declarations do not pose a problem.

to be taken. One possible approach (see Chapter 6) is to axiomatize *collections* [79] of access objects, and use an approach based on algebraic specification checking (see Chapter 4).

3.5.5 Transformation of Exception Annotations

The Anna Transformer restricts the use of propagation annotations in the following manner: After the transformations described in Section 3.3, all propagation annotations must occur before any other declaration or annotation within a declarative region. Hence, after the transformations of Section 3.3, all *frames*[8] have the following structure:

```
frame_header
    all propagation annotations
    rest of declarative region
begin
    sequence_of_statements
[exception
    exception_handlers]
end;
```

The first step in the transformation creates a new Ada *block* as shown below:

```
frame_header
   all propagation annotations
begin
   declare
         (1)
      rest of declarative region
   begin
      sequence_of_statements
  [exception
      exception_handlers]
   end;
exception
      (2)
end;
```

[8] A frame in Ada is either a block statement or the body of a subprogram, package, task unit, or generic unit.

The inner block has the same semantics as the original frame. The function of the outer frame is to monitor both normal and abnormal exits from the inner block at the same time maintain the external appearance of the frame just as it was before the transformation. In the next step of the transformation, the propagation annotations will be transformed into *out* annotations (which will be placed at location 1), and checks (which will be placed at location 2). The transformation of *out* annotations are described in the next section.

Strong Propagation Annotations

Strong propagation annotations have the form:

```
C => raise E;
```

The condition C is evaluated during elaboration of the annotation and assigned to a variable, just as in the case of initial expressions. When this variable is TRUE, exit from the block has to take place abnormally by propagating the exception E. Hence, checks are generated at all other exit points to ensure that this variable is not TRUE at these points. This is done as shown below:

```
Before:
frame_header
   --| C => raise E;
begin
   declare
      rest of declarative region
   begin
      sequence_of_statements
  [exception
      exception_handlers]
   end;
exception
   ...
end;
```

```
After:
frame_header
   C1 : BOOLEAN := C;
begin
   declare
      --| out (not C1);
      rest of declarative region
   begin
      sequence_of_statements
  [exception
      exception_handlers]
   end;
exception
   when E =>
      raise;
   when x =>
      -- For all exceptions, x, not equal to E
      CHECK(not C1);
      raise;
end;
```

Weak Propagation Annotations

There are two forms of weak propagation annotations. The first, has the form:

```
raise E1 | . . | En;
```

This means that it is possible to leave the scope of the annotation abnormally by raising the exceptions $E_1, \ldots, E_n$. There is no checking necessary for this kind of annotation, for it does not restrict the underlying program in any way. It can be considered a form of formal documentation and some static semantic analysis could be performed based on it.

The other form of weak propagation annotation is:

```
raise E1 | ... | En => C;
```

This is equivalent to a list of n weak propagation annotations, one for each exception. The transformations are performed for each of the n exceptions individually. For the rest of this section, assume the weak propagation annotation being transformed is:

```
raise E => C;
```

A check is generated in the exception handler for E to ensure that C does in fact hold at that point. This is done as shown below:

```
Before:
frame_header
   --| raise E => C;
begin
   declare
      rest of declarative region
   begin
      sequence_of_statements
  [exception
      exception_handlers]
   end;
exception
   ...
end;

After:
frame_header
begin
   declare
      rest of declarative region
   begin
      sequence_of_statements
  [exception
      exception_handlers]
   end;
```

```
exception
   when E =>
      CHECK(C);
   ...
end;
```

Example 3.26:

Before:

```
-- This is slightly different from the OPEN defined in the pre-
-- defined package TEXT_IO.
procedure OPEN(FILE:in out FILE_TYPE;NAME:in STRING) is
   --| not EXISTS(NAME) => raise NAME_ERROR;
   --| IS_OPEN(NAME) => raise STATUS_ERROR;
   --| raise STATUS_ERROR => IS_OPEN(NAME);
   ...
begin
   ...
end OPEN;
```

After:

```
procedure OPEN(FILE:in out FILE_TYPE;NAME:in STRING) is
   C1:BOOLEAN := not EXISTS(NAME);
   C2:BOOLEAN := IS_OPEN(NAME);
   -- Note: C1 and C2 capture the values of the expressions on
   -- entry into OPEN. These expressions may change value
   -- during the execution of OPEN, but C1 and C2 will not
   -- change value.
begin
   declare
      --| out (not C1);
      --| out (not C2);
      ...
   begin
      ...
```

```
    end;
exception
    when NAME_ERROR =>
        CHECK(not C2);
        raise;
    when STATUS_ERROR =>
        CHECK(not C1);
        CHECK(IS_OPEN(NAME));
        raise;
    when others =>
        CHECK(not C1);
        CHECK(not C2);
        raise;
end OPEN;
```

3.5.6 Transformation of Out Annotations

Out annotations are constraints on the normal exit states of their scopes. After the transformations described in Section 3.3, all annotations will occur as declarative annotations. Hence the scope of *out* annotations is either a block statement, the body of a subprogram, package, task unit or generic unit. *Out* annotations have to be checked at all normal exit points within the scope of the annotation. There are five different kinds of exit points:

1. The end of the sequence of statements
2. The end of the sequence of statements of an exception handler
3. *Return* statements
4. *Exit* statements
5. *Goto* statements

At each of these exit points, the list of all *out* annotations to be tested are determined. This list consists of all *out* annotations in whose scope the exit point occurs, but not including those which include in their scope the point to where control is transferred. Each of these annotations is tested by calling the procedure CHECK in the Anna runtime library.

Example 3.27:

Before:

```
procedure FOO is
   --| out F1;
   ...
begin
   declare
      --| out F2;
      ...
   begin
      loop
         declare
            --| out F3;
            ...
         begin
            ...
            exit;
            ...
            return;
            ...
            goto L;
            ...
         end;
      end loop;
      ...
      return;
      ...
   end;
   ...
<<L>>
   ...
   goto L;
   ...
exception
   when others =>
      ...
end FOO;
```

After:

```
procedure FOO is
   ...
begin
   declare
      ...
   begin
      loop
         declare
            ...
         begin
            ...
            CHECK(F3);
            exit;
            ...
            CHECK(F1); CHECK(F2); CHECK(F3);
            return;
            ...
            CHECK(F2); CHECK(F3);
            goto L;
            ...
            CHECK(F3);
         end;
      end loop;
      ...
      CHECK(F1); CHECK(F2);
      return;
      ...
      CHECK(F2);
   end;
   ...
<<L>>
   ...
   -- No checks required here
   goto L;
   ...
```

```
    CHECK(F1);
exception
    when others =>
        ...
        CHECK(F1);
end FOO;
```

There is one situation which is not covered in the above framework. This is the case of returning from a function. In this case, the *out* annotations have to be checked *after* the return expression has been evaluated. This problem is tackled by generating a function whose only function is to check the relevant *out* annotations as a side effect.

Example 3.28:

Before:

```
function C(N,R:INTEGER) return INTEGER is
    NUMERATOR,FACT_R:INTEGER;
    --| out (FACT_R = FACTORIAL(R));
begin
    ...
    return NUMERATOR/FACT_R;
    -- Out annotation has to be tested after evaluation of NU-
    -- MERATOR/FACT_R
end C;
```

After:

```
function C(N,R:INTEGER) return INTEGER is
    NUMERATOR,FACT_R:INTEGER;
begin
    ...
    declare
        function OUT_CHECK(X:INTEGER) return INTEGER is
        begin
            CHECK_EXP(FACT_R = FACTORIAL(R));
            return X;
        end OUT_CHECK;
    begin
```

```
        return OUT_CHECK(NUMERATOR/FACT_R);
    end;
end C;
```

3.6 Concurrent Checking of Generalized Assertions

The Anna Transformer has an option using which checking tasks are generated instead of checking functions. These tasks execute concurrently with the underlying program. However, this approach has the disadvantage that the underlying program may execute for some time in an inconsistent state. *Check-points* can be defined at various places where the underlying program can be forced to wait until specified checks are completed. The details of these transformation and checking methodology are described in [81].

Chapter 4

Algebraic Specification Checking

In Chapter 1, an example was presented to motivate algebraic specification checking. A general discussion of the principles of consistency checking was presented in Chapter 2. In this chapter the details of algebraic specification checking are presented.

4.1 Abstract Data Types and Algebraic Specifications

An *abstract data type* defines a set of *data objects*, and a set of *operations* on these data objects. These are encapsulated in such a way that the user of the abstract data type can only manipulate the data objects by using the operations provided.

The Ada package is an encapsulation unit using which abstract data types can be defined. The data objects are defined by Ada type declarations, and the abstract operations are defined as Ada subprograms. Example 1.2 in Chapter 1 defines the DEQUE abstract data type using the Ada package. There are other ways in which an abstract data type can be defined using Ada packages. For example, procedures could replace the functions; and the data objects could be defined implicitly by the state of the package. However, for the purposes of this thesis, an abstract data type is assumed to be implemented as an Ada package with one type declaration. This package may refer to other type declarations, or other abstract data types. These types are referred to as the *auxiliary* types. This package contains functions that operate on the package type and auxiliary types and return values of these types, but does not contain any procedures. A simple transformation can convert any arbitrary implementation of an abstract data type to the kind of implementation stated above.

As mentioned in Chapter 1, for the purpose of this thesis, an algebraic specification of an abstract data type is defined as a set of equations whose terms comprise of the abstract operations and variables that are universally quantified over the domain of abstract objects. The Anna package axioms in Example 1.2 in Chapter 1 form an algebraic specification of the DEQUE abstract data type.

4.2 Algebraic Specification Checking

Algebraic specification checking is the task of checking the implementation of an abstract data type (the package body in Ada) against the algebraic specification of the abstract data type. First, however, one has to define what kinds of checks are involved in this process. This is not as obvious as in the case of checking against generalized assertions.

At any point during the execution of a program that uses an abstract data type, there is a set of objects declared to be of the abstract data type. Some of these objects will have been assigned values from the domain of the abstract data type. Since the only way to compute values of the abstract data type is to execute the abstract data type operations, all values that have been assigned to the above-mentioned objects are a result of applying a composition of these abstract data type operations to the constants[1] of the abstract data type. *That is, for each value of the abstract data type computed, there is a corresponding term comprising only of operations and constants of the abstract data type whose evaluation results in this value.* Let Θ be the set of all such terms at any particular point during the execution of the program. Obviously, Θ grows in size as the program continues to execute. The set Θ after the execution of the first set of assignment statements of Example 1.2 in Chapter 1 contains:

CREATE
LEFT_PUSH(CREATE, E_0)
LEFT_TOP(LEFT_PUSH(CREATE, E_0))
LEFT_POP(LEFT_PUSH(CREATE, E_0))

and after the execution of the second set of statements in the same example, Θ contains:

[1] A constant is an operation that does not take any parameters.

CREATE
LEFT_PUSH(CREATE, E_2)
RIGHT_PUSH(LEFT_PUSH(CREATE, E_2), E_3)
LEFT_POP(RIGHT_PUSH(LEFT_PUSH(CREATE, E_2), E_3))
RIGHT_POP(LEFT_POP(RIGHT_PUSH(LEFT_PUSH(CREATE, E_2), E_3)))

The following definition of algebraic specification checking is the basis of the methodologies and algorithms discussed in this chapter:

> *Algebraic specification checking involves comparing the terms in* Θ *with each other. If any two terms can be proved equal based on the algebraic specification, then the corresponding program values have to be compared with each other. If they are not equal, then the program has become inconsistent with the algebraic specification.*

In Example 1.1, since LEFT_POP(LEFT_PUSH(CREATE, E_0)) can be proved to be equal to CREATE, therefore, the corresponding program values, namely D_2 and D_0, have to compared with each other.

There are some situations which the above definition of algebraic specification checking does not cover. That is, it is possible for a program to reach an inconsistent state with respect to an algebraic specification and yet an *ideal* algebraic specification checking system (based on the above definition of algebraic specification checking) will not able to detect this inconsistency. The following example illustrates such a situation:

Example 4.1: Consider the following sequence of statements that execute operations of the DEQUE abstract data type of Example 1.2:

```
D1 := LEFT_PUSH(D0, E0);
D3 := LEFT_POP(D2);
if D1 = D2 then
    -- (1)
    ...
end if;
```

At location 1 it can be shown that D_3 has to be equal to D_0 (since $D_1 = D_2$). However, an ideal algebraic specification checking system will not be able to determine this. This is

because, though $D_1 = D_2$ is true at this location, the algebraic specification checking model does not make use of it. Extending the model of algebraic specification checking of this thesis to cover such situations is a topic for future research.

As has been mentioned in Chapter 2, a theorem prover forms part of the algebraic specification checking system. In this chapter, a specialized theorem prover, called the Chromatic Theorem Prover is described. However, other similar provers can also be used. When the program starts executing, the theorem prover is initialized with the algebraic specifications. After this, a call is made to the theorem prover every time an operation of the abstract data type is executed.

The theorem prover is an Ada generic package. The body of this package incorporates the specific theorem proving algorithms. The *generic package specification is not influenced* in any way by the underlying algorithms. A simplified version of the generic package specification is shown below:

```
generic

    type ABSTRACT_TYPE is limited private;
    with procedure ASSIGN_SLNO(A:in out ABSTRACT_TYPE;SLNO:INTEGER);
    with function GET_SLNO(A:ABSTRACT_TYPE) return INTEGER;
    with function EQ'X,Y:ABSTRACT_TYPE) return BOOLEAN;
    with procedure COPY(X:ABSTRACT_TYPE;Y:out ABSTRACT_TYPE);
    type AUXILIARY_TYPE is limited private;
    with function EQ(X,Y:AUXILIARY_TYPE) return BOOLEAN;
    AXIOM_FILE:STRING;

package AXIOM_CHECKER is

    procedure OP_OVER(OP:CHARACTER;
                      INPUT:ABSTRACT_TYPE;
                      OUTPUT:in out ABSTRACT_TYPE);
```

```
        procedure OP_OVER(OP:CHARACTER;
                          AUX:AUXILIARY_TYPE;
                          INPUT:ABSTRACT_TYPE;
                          OUTPUT:in out ABSTRACT_TYPE);

    end AXIOM_CHECKER;
```

The Anna Transformer maps each of the abstract operations to a character. This mapping corresponds to the isomorphism function that will be introduced later in this chapter. A typical mapping in the case of Example 1.2 is:

CREATE	→	'a'
LEFT_PUSH	→	'b'
LEFT_POP	→	'c'
LEFT_TOP	→	'd'
RIGHT_PUSH	→	'e'
RIGHT_POP	→	'f'
RIGHT_TOP	→	'g'

The generic package exports a family of procedures called OP_OVER. The appropriate procedure from this family is called at the end of the execution of each abstract operation. The calls to OP_OVER provides the theorem prover with the information necessary to update the set Θ that it maintains. In the above package, two such procedures are shown. The first has three parameters. OP corresponds to the abstract operation; INPUT to the input parameter of the operation and OUTPUT to the result of the operation. This procedure is called by abstract operations which take one abstract data type parameter and returns an abstract data type value. In Example 1.2, the functions LEFT_POP and RIGHT_POP call this procedure. The second procedure has an additional parameter which is of the auxiliary type. This is called by operations with a parameter of the auxiliary type in addition to the parameter of the abstract data type. In the case of the DEQUE package, the auxiliary type is ELEMENT. This procedure is called by the functions LEFT_PUSH and RIGHT_PUSH.

The Anna Transformer performs a transformation on the abstract data type so that a serial number can be associated with it. One way to do this is to create a record structure with two components—one the original abstract data type and the other the serial number. The serial number is assigned and used by the theorem prover only. Since the theorem

prover performs only symbolic manipulations with the abstract objects, the serial numbers act as the symbolic representation of the objects. Hence two operations are provided to the theorem prover as generic formal parameters for this purpose: ASSIGN_SLNO and GET_SLNO.

The algebraic specification is copied (after transforming the operations into characters) into a file. This file is read in by the theorem prover during its initialization. The location of the file is provided by the generic formal parameter AXIOM_FILE.

Whenever a new variable is encountered by the theorem prover, it makes a copy of it. It is with this copy that any future comparisons for equality are made. The generic formal parameters EQ (over ABSTRACT_TYPE) and COPY are provided for this purpose. It may be necessary for the programmer to explicitly provide these two operations in packages where these operations are not already present.

The DEQUE package of Example 1.2 after transformation

```
package DEQUE_PACKAGE is
   -- The following modification to the type definition is to in-
   -- clude the serial number. In the case of a private type,
   -- this transformation is performed in the full type defini-
   -- tion in the private part.
   type DEQUE is record
      ...;
      SLNO:INTEGER;
   end record;
   function CREATE return DEQUE;
   function LEFT_PUSH(D:DEQUE;E:ELEMENT) return DEQUE;
   function LEFT_POP(D:DEQUE) return DEQUE;
   function LEFT_TOP(D:DEQUE) return ELEMENT;
   function RIGHT_PUSH(D:DEQUE;E:ELEMENT) return DEQUE;
   function RIGHT_POP(D:DEQUE) return DEQUE;
```

```
    function RIGHT_TOP(D:DEQUE) return ELEMENT;
end DEQUE_PACKAGE;

package body DEQUE_PACKAGE is

    -- The following two subprograms are provided by the user.

    function EQUALS(D1,D2:DEQUE) return BOOLEAN is
    ...
    end EQUALS;

    procedure COPY(D1:DEQUE;D2:in out DEQUE) is
    ...
    end COPY;

    -- The following two subprograms and the generic instanti-
    -- ation are generated by the Anna Transformer.

    procedure ASSIGN_SLNO(A:in out DEQUE;SLNO:INTEGER) is
    begin
        A.SLNO := SLNO;
    end ASSIGN_SLNO;

    function GET_SLNO(A:DEQUE) return INTEGER is
    begin
        return A.SLNO;
    end GET_SLNO;

    package DEQUE_AXIOM_CHECKER is new AXIOM_CHECKER
        (DEQUE,
         ASSIGN_SLNO,
         GET_SLNO,
         EQUALS,
         COPY,
         ELEMENT,
         "=",
         "deque.ax");
```

```
function LEFT_PUSH(D:DEQUE;E:ELEMENT) return DEQUE is
   ...
begin
   ...
   -- A return point
   OP_OVER('b',E,D,D');
   return D';
   ...
end LEFT_PUSH;

:

end DEQUE_PACKAGE;
```

The rest of this chapter focuses on the theorem prover. The complexity of the general problem of (the theorem proving aspects of) algebraic specification checking is discussed followed by some background material. The Chromatic Theorem Prover is then discussed, followed by a description of how the theorem proving operations can be implemented incrementally.

4.3 Complexity of the General Problem

In this section, the problem of algebraic specification checking is shown to be at least as complex (computationally) as the *word problem.*

Given an algebra A[2] specified by a set of equations Σ, and given two terms t_1 and t_2 from A, the *word problem* is the process of deciding whether or not the validity of $t_1 = t_2$ follows from Σ ($\Sigma \models t_1 = t_2$).

For any algebra A, there is a corresponding abstract data type. Corresponding to the equations Σ that specify the algebra, there is an algebraic specification that specifies the abstract data type. Consider any two terms t_1 and t_2 from A. It is possible for the program using the abstract data type to evaluate the corresponding two terms of the abstract data type. If it does evaluate these terms and if $\Sigma \models t_1 = t_2$, then the program must perform a check to ensure that the values of these terms are equal to each other. However, if $\Sigma \not\models t_1 = t_2$, the program must not perform this check. The process of deciding whether or

[2] An algebra is a domain of values together with a set of operations on these values.

not this check must be performed therefore involves deciding whether or not $\Sigma \models t_1 = t_2$ and hence, algebraic specification checking is at least as difficult as solving word problems.

Since word problems are in general undecidable, therefore algebraic specification checking is also, in general, undecidable. Hence, the theorem proving operations can only be performed partially. The Knuth-Bendix algorithm [59] is one already existing algorithm that can be used to perform the theorem proving operations necessary. The following sections describe a *new* theorem proving algorithm which has been designed specifically for the purpose of algebraic specification checking. This new algorithm focuses on real-life abstract data types, and hence works on a smaller subset than many other theorem provers. However, this trade-off results in an improvement in the performance of the algorithm. When implemented incrementally[3], the time complexity of this algorithm does not change as the program generates more and more terms, although the space required increases as the program continues to run.

The subset for which this algorithm works is characterized as follows[4]:

1. The operations of the abstract data type have to be unary,

2. There is exactly one constant of the abstract data type[5], and

3. The algebraic specifications have to be of the form $t_1(x) = t_2(x)$[6].

The reason for the choice of this subset is that many real-life abstract data types like stacks, queues, sets and symbol tables are either already in the subset or can be transformed in a simple manner to fit into the subset. The mathematical entity that is used to model this subset is the *Thue system* [116][7].

The next section describes Thue systems and how abstract data types and their algebraic specifications within the above subset can be matched to them. A few examples (also in the next section) illustrate that a large number of real-life abstract data types fit into this subset.

---

[3] The nature of this incrementality is discussed in Section 4.6.

[4] This characterization will be modified slightly later.

[5] More than one constant creates disjoint problem spaces as will be seen later. Hence there is no loss of generality by assuming just one constant.

[6] Note that this disallows any specification that relates constants of the abstract data type.

[7] The author is indebted to Deepak Kapur for introducing him to Thue Systems.

4.4 Thue Systems

Thue systems are rewriting systems specified using equations over strings where the concatenation operation satisfies the associativity property. Let Σ be a finite alphabet. Σ^* is the set of all finite strings over Σ. Let the empty string be denoted by λ. λ is an identity of the concatenation operation (i.e., $\lambda x = x\lambda = x$). A *Thue system* T is a binary relation on Σ^*. Every element of T is called an *equation* of T. An example of a Thue system which describes sequences of the unary operators + and − is shown below:

Example 4.2: Let $\Sigma = \{+, -\}$. Therefore Σ^* is the set of all finite sequences of these unary operators. Let $T = \{< +, \lambda >, < --, \lambda >\}$. Then T is a Thue system that describes how a sequence of these unary operations can be rewritten equivalently.

The rest of this section is devoted to showing how certain abstract data types and their algebraic specifications can be matched to Thue systems. First some background material is overviewed, and a few central theorems are stated and proved.

4.4.1 An Overview of Proof Theory in Equational Logic

Let Γ be a set of equations and $s = t$ be an equation (in an algebra A). Γ *logically implies* $s = t$ ($\Gamma \models s = t$) if and only if every *model* that satisfies all equations in Γ also satisfies $s = t$. A simple methodology is desirable to deduce logical implication, and for this a *proof theory* has been developed. Γ proves $s = t$ ($\Gamma \vdash s = t$) if and only if there is a finite sequence of equations $s_1 = t_1, \ldots, s_n = t_n$ such that:

- the last equation is $s = t$, and
- each $s_i = t_i$ is either a member of Γ, or of the form $t = t$, or it follows from the previous equations according to one of the following rules of inference:
  1. *symmetry:* From $s_i = t_i$ infer $t_i = s_i$
  2. *transitivity:* From $s_i = s_j$ and $s_j = s_k$ infer $s_i = s_k$
  3. *composition:* From $t_1 = t'_1, \ldots, t_k = t'_k$ infer $f(t_1, \ldots, t_k) = f(t'_1, \ldots, t'_k)$
  4. *substitution:* Let t_e^x denote the substitution of all free occurrences of the variable x by the expression e. From $s_i = s_j$, infer $s_{i_e}^x = s_{j_e}^x$.

Theorem 4.1 (Birkhoff) *Let Γ be a set of equations and $s = t$ be an equation. Then $\Gamma \vdash s = t$ if and only if $\Gamma \models s = t$.*

For a proof of this theorem, please refer to either [9] or Page 95 of [14].

4.4.2 Terminology, Definitions and Lemmas

Define a binary relation $\leftrightarrow_T$ on Σ^* as follows: for any u, v, x, y in Σ^* such that either $\langle u, v\rangle$ is in T or $\langle v, u\rangle$ is in T, $xuy \leftrightarrow_T xvy$. The *Thue congruence* generated by T, $\leftrightarrow_T^*$, is the reflexive transitive closure of the relation $\leftrightarrow_T$. Since the relation $\leftrightarrow_T$ is symmetric, therefore $\leftrightarrow_T^*$ is an equivalence relation. Deducing whether or not $w \leftrightarrow_T^* z$ for any two strings w, z in Σ^* is an undecidable problem. This was shown by Post [97] and independently by Markov [84].

Let A be an abstract data type with only unary operations $f_1, \ldots, f_n$ and one constant c. Consider the terms of A. Since all the operations are unary, the terms can only be a repeated application of these operations on some variable or on c. Define $\mathcal{T}(x)$ to be the set of all terms of A where the operations are applied on x. Therefore each member of $\mathcal{T}(x)$ will be of the form $f_{i_1}(\ldots(f_{i_m}(x))\ldots)$ where $m \geq 0$ and $1 \leq i_1, \ldots, i_m \leq n$.

Let S be the algebraic specification of A where each equation in S is of the form $t_1(x) = t_2(x)$ where x is some variable, and $t_1(x), t_2(x)$ are in $\mathcal{T}(x)$. In other words, each equation of S is restricted to have the same (implicitly) universally quantified variable on both sides.

Let $\Sigma = \{e_1, \ldots, e_n\}$ be an alphabet and let Σ^* be the set of all finite strings over Σ. Define an *isomorphism* $\mathcal{I}_x$[8] from $\mathcal{T}(x)$ to Σ^* as follows: $\mathcal{I}_x(x) = \lambda$, and for all $t(x)$ in $\mathcal{T}(x)$, $\mathcal{I}_x(f_i(t(x))) = e_i\mathcal{I}_x(t(x))$, $1 \leq i \leq n$. For example, $\mathcal{I}_x(f_1(f_2(f_2(x)))) = e_1e_2e_2$. $\mathcal{I}_x^{-1}$ is the inverse of $\mathcal{I}_x$. Let T be a binary relation on Σ^* defined as follows: For each equation $t_1(x) = t_2(x)$ in S, $\langle \mathcal{I}_x(t_1(x)), \mathcal{I}_x(t_2(x))\rangle$ is a member of T. Nothing else is a member of T.

Lemma 4.2 *If $x, y, z, w \in \Sigma^*$ and $x \leftrightarrow_T^* y$, then $wxz \leftrightarrow_T^* wyz$.*

Proof Since $\leftrightarrow_T^*$ is the reflexive transitive closure of $\leftrightarrow_T$, there are two cases to consider:

1. There exists $a_0, \ldots, a_n$ in Σ^* where $n > 0$ such that $x = a_0$, $y = a_n$, and $a_i \leftrightarrow_T a_{i+1}$, $0 \leq i < n$:

   Consider $a_i \leftrightarrow_T a_{i+1}$ for any i. From the definition of $\leftrightarrow_T$, there exists u, v, p, q in Σ^* such that $a_i = puq$ and $a_{i+1} = pvq$ and either $\langle u, v\rangle$ is in T or $\langle v, u\rangle$ is in T. Let $r = wp$ and $s = qz$. Again, from the definition of $\leftrightarrow_T$, $rus \leftrightarrow_T rvs$. Since

[8] The mapping of operations to characters described in Section 4.2 is the implementation of this isomorphism function.

$rus = wpuqz = wa_iz$ and $rvs = wpvqz = wa_{i+1}z$, therefore $wa_iz \leftrightarrow_T wa_{i+1}z$. Also since $\leftrightarrow_T^*$ is the reflexive transitive closure of $\leftrightarrow_T$, $wa_0z \leftrightarrow_T^* wa_nz$. Hence, $wxz \leftrightarrow_T^* wyz$.

2. $x = y$:

   Then $wxz = wyz$, and since $\leftrightarrow_T^*$ is the reflexive transitive closure of $\leftrightarrow_T$, $wxz \leftrightarrow_T^* wyz$.

□

Lemma 4.3 *If $t_1(x), t_2(x)$ are in $\mathcal{T}(x)$, then $\mathcal{I}_x(t_1(t_2(x))) = \mathcal{I}_x(t_1(x))\mathcal{I}_x(t_2(x))$.*

Proof This is proved by induction on the number of operations in $t_1(x)$.
Base case: The number of operations in $t_1(x)$ is 0, or $t_1(x) = x$. Therefore $t_1(t_2(x)) = t_2(x)$ and hence $\mathcal{I}_x(t_1(t_2(x))) = \mathcal{I}_x(t_2(x)) = \lambda\mathcal{I}_x(t_2(x)) = \mathcal{I}_x(t_1(x))\mathcal{I}_x(t_2(x))$.
Inductive step: Assume $\mathcal{I}_x(t_1(t_2(x))) = \mathcal{I}_x(t_1(x))\mathcal{I}_x(t_2(x))$ for all $t_1(x)$ with less than m operations. Now consider any $t_1(x)$ with exactly m operations. Then $t_1(x)$ is of the form $f_i(t_1'(x))$ where $t_1'(x)$ has exactly $m-1$ operations. Therefore $\mathcal{I}_x(t_1(t_2(x))) = \mathcal{I}_x(f_i(t_1'(t_2(x)))) = e_i\mathcal{I}_x(t_1'(t_2(x))) = e_i\mathcal{I}_x(t_1'(x))\mathcal{I}_x(t_2(x)) = \mathcal{I}_x(f_i(t_1'(x)))\mathcal{I}_x(t_2(x)) = \mathcal{I}_x(t_1(x))\mathcal{I}_x(t_2(x))$. □

Lemma 4.4 *If u is in Σ^*, then $\mathcal{I}_x^{-1}(ue_i) = \mathcal{I}_{f_i(x)}^{-1}(u)$.*

Proof This is proved by induction on the length of u.
Base case: The length of u is 0, or $u = \lambda$. Therefore $\mathcal{I}_x^{-1}(ue_i) = \mathcal{I}_x^{-1}(e_i) = f_i(x) = \mathcal{I}_{f_i(x)}^{-1}(\lambda) = \mathcal{I}_{f_i(x)}^{-1}(u)$.
Inductive step: Assume $\mathcal{I}_x^{-1}(ue_i) = \mathcal{I}_{f_i(x)}^{-1}(u)$ for all u with length less than m. Now consider any u with length m. Then $u = e_ju'$ where u' has length $m-1$. Therefore $\mathcal{I}_x^{-1}(ue_i) = \mathcal{I}_x^{-1}(e_ju'e_i) = f_j(\mathcal{I}_x^{-1}(u'e_i)) = f_j(\mathcal{I}_{f_i(x)}^{-1}(u')) = \mathcal{I}_{f_i(x)}^{-1}(e_ju') = \mathcal{I}_{f_i(x)}^{-1}(u)$. □

Lemma 4.5 *If u, v are in Σ^*, then $\mathcal{I}_x^{-1}(uv)$ is $\mathcal{I}_x^{-1}(u)$ with $\mathcal{I}_x^{-1}(v)$ substituted for x. Or in other words, $\mathcal{I}_x^{-1}(uv) = \mathcal{I}_{\mathcal{I}_x^{-1}(v)}^{-1}(u)$.*

Proof This is proved using induction on the length of v.
Base case: The length of v is 0, or $v = \lambda$. Therefore $\mathcal{I}_x^{-1}(v) = x$ and $\mathcal{I}_x^{-1}(uv) = \mathcal{I}_x^{-1}(u) = \mathcal{I}_{\mathcal{I}_x^{-1}(v)}^{-1}(u)$.
Inductive step: Assume $\mathcal{I}_x^{-1}(uv) = \mathcal{I}_{\mathcal{I}_x^{-1}(v)}^{-1}(u)$ for all v with length less than m. Now consider any v with length m. Then $v = v'e_i$ where v' has length $m-1$. Using the

inductive hypothesis and lemma 4.4, $\mathcal{I}_x^{-1}(uv) = \mathcal{I}_x^{-1}(uv'e_i) = \mathcal{I}_{f_i(x)}^{-1}(uv') = \mathcal{I}_{\mathcal{I}_{f_i(x)}^{-1}(v')}^{-1}(u) = \mathcal{I}_{\mathcal{I}_x^{-1}(v'e_i)}^{-1}(u) = \mathcal{I}_{\mathcal{I}_x^{-1}(v)}^{-1}(u)$. □

Corollary 4.6 *If u, v, q are in Σ^* and $S \vdash \mathcal{I}_x^{-1}(u) = \mathcal{I}_x^{-1}(v)$, then $S \vdash \mathcal{I}_x^{-1}(uq) = \mathcal{I}_x^{-1}(vq)$.*

Lemma 4.7 *If p, u, v are in Σ^* and $S \vdash \mathcal{I}_x^{-1}(u) = \mathcal{I}_x^{-1}(v)$, then $S \vdash \mathcal{I}_x^{-1}(pu) = \mathcal{I}_x^{-1}(pv)$.*

Proof This is proved using induction on the length of p.
Base case: The length of p is 0, or $p = \lambda$. Therefore $\mathcal{I}_x^{-1}(pu) = \mathcal{I}_x^{-1}(u)$ and $\mathcal{I}_x^{-1}(pv) = \mathcal{I}_x^{-1}(v)$. Hence, $S \vdash \mathcal{I}_x^{-1}(pu) = \mathcal{I}_x^{-1}(pv)$.
Inductive step: Assume $S \vdash \mathcal{I}_x^{-1}(pu) = \mathcal{I}_x^{-1}(pv)$ for all p with length less than m. Now consider any p with length m. Then $p = e_i p'$ where p' has length $m - 1$. Therefore $\mathcal{I}_x^{-1}(pu) = \mathcal{I}_x^{-1}(e_i p'u) = f_i(\mathcal{I}_x^{-1}(p'u))$ and $\mathcal{I}_x^{-1}(pv) = \mathcal{I}_x^{-1}(e_i p'v) = f_i(\mathcal{I}_x^{-1}(p'v))$. From the induction hypothesis, $S \vdash \mathcal{I}_x^{-1}(p'u) = \mathcal{I}_x^{-1}(p'v)$ and therefore (using the rule of composition) $S \vdash f_i(\mathcal{I}_x^{-1}(p'u)) = f_i(\mathcal{I}_x^{-1}(p'v))$, and hence $S \vdash \mathcal{I}_x^{-1}(pu) = \mathcal{I}_x^{-1}(pv)$. □

Lemma 4.8 *Let $s_1(x) = t_1(x), \ldots, s_n(x) = t_n(x)$ be a sequence of equations in a proof in A. Then $\mathcal{I}_x(s_i(x)) \leftrightarrow_T^* \mathcal{I}_x(t_i(x))$, $1 \leq i \leq n$.*

Proof The lemma follows from the proof by induction on j that $\mathcal{I}_x(s_i(x)) \leftrightarrow_T^* \mathcal{I}_x(t_i(x))$, $1 \leq i \leq j$.
Base case: $j = 0$. In this case, the result holds trivially.
Inductive step: Assume that $\mathcal{I}_x(s_i(x)) \leftrightarrow_T^* \mathcal{I}_x(t_i(x))$, $1 \leq i < j$. Now consider $s_j(x) = t_j(x)$. This equation falls into one of the following categories:

1. $s_j(x) = t_j(x)$ is a member of S. In this case, obviously $\mathcal{I}_x(s_j(x)) \leftrightarrow_T^* \mathcal{I}_x(t_j(x))$.

2. $s_j(x) = t_j(x)$ is of the form $t = t$, or follows from one of the previous equations by *symmetry* or *transitivity*. Since $\leftrightarrow_T^*$ is an equivalence relation, therefore $\mathcal{I}_x(s_j(x)) \leftrightarrow_T^* \mathcal{I}_x(t_j(x))$.

3. $s_j(x) = t_j(x)$ is of the form $f_k(s_i(x)) = f_k(t_i(x))$ for some $i < j$. Then $\mathcal{I}_x(s_j(x)) = \mathcal{I}_x(f_k(s_i(x))) = e_k\mathcal{I}_x(s_i(x))$ and $\mathcal{I}_x(t_j(x)) = \mathcal{I}_x(f_k(t_i(x))) = e_k\mathcal{I}_x(t_i(x))$. From the inductive hypothesis and lemma 4.2, $e_k\mathcal{I}_x(s_i(x)) \leftrightarrow_T^* e_k\mathcal{I}_x(t_i(x))$ and therefore $\mathcal{I}_x(s_j(x)) \leftrightarrow_T^* \mathcal{I}_x(t_j(x))$.

4. $s_j(x) = t_j(x)$ is of the form $s_i(u(x)) = t_i(u(x))$ for some $i < j$. Then from lemma 4.3, $\mathcal{I}_x(s_i(u(x))) = \mathcal{I}_x(s_i(x))\mathcal{I}_x(u(x))$ and $\mathcal{I}_x(t_i(u(x))) = \mathcal{I}_x(t_i(x))\mathcal{I}_x(u(x))$. From the inductive hypothesis and lemma 4.2, $\mathcal{I}_x(s_i(x))\mathcal{I}_x(u(x)) \rightarrow_T^* \mathcal{I}_x(t_i(x))\mathcal{I}_x(u(x))$ and therefore $\mathcal{I}_x(s_j(x)) \rightarrow_T^* \mathcal{I}_x(t_j(x))$. □

4.4.3 Matching Abstract Data Types to Thue Systems

The isomorphism $\mathcal{I}_x$ defined in the previous section is the method used to match abstract data types and their algebraic specifications to Thue systems.

Example 4.3: This is an example of the matching process. The following is a package that defines an abstract data type with the unary functions PLUS and MINUS. The algebraic specification within this package describes these functions.

```
package NUMBER_PACKAGE is
   type NUMBER is private;
   function PLUS(N:NUMBER) return NUMBER;
   function MINUS(N:NUMBER) return NUMBER;
--| axiom
--|    for all N:NUMBER =>
--|       PLUS(N) = N,
--|       MINUS(MINUS(N)) = N;
end NUMBER_PACKAGE;
```

By choosing $\mathcal{I}_x(\text{PLUS}(x)) = +$ and $\mathcal{I}_x(\text{MINUS}(x)) = -$, the above abstract data type and its algebraic specification can be matched to the Thue system of Example 4.2.

The motivation for performing this matching process is to derive theorems about abstract data types by performing proofs on the corresponding Thue system. For this approach to be useful, an effective *proof system* is required for Thue systems. The following theorem lays the groundwork for this proof system:

Theorem 4.9 *If $s(x), t(x)$ are terms from $T(x)$, then $S \vdash s(x) = t(x)$ if and only if $\mathcal{I}_x(s(x)) \rightarrow_T^* \mathcal{I}_x(t(x))$.*

Proof From lemma 4.8, if $S \vdash s(x) = t(x)$ then $\mathcal{I}_x(s(x)) \rightarrow_T^* \mathcal{I}_x(t(x))$. Also, from corollary 4.5 and lemma 4.7, if p, u, v, q are in Σ^* and either $\langle u, v\rangle$ is in T or $\langle v, u\rangle$ is in T, then $S \vdash \mathcal{I}_x^{-1}(u) = \mathcal{I}_x^{-1}(v)$, and therefore $S \vdash \mathcal{I}_x^{-1}(puq) = \mathcal{I}_x^{-1}(pvq)$. Hence for any w, z in Σ^*,

if $w \rightarrow_T z$, then $S \vdash \mathcal{I}_x^{-1}(w) = \mathcal{I}_x^{-1}(z)$, and since $\rightarrow_T^*$ is the reflexive transitive closure of $\rightarrow_T$, therefore if $w \rightarrow_T^* z$, then $S \vdash \mathcal{I}_x^{-1}(w) = \mathcal{I}_x^{-1}(z)$. □

Corollary 4.10 (Proof System for Thue Systems) *If $s(x), t(x)$ are terms from $\mathcal{T}(x)$, then $S \vdash s(x) = t(x)$ if and only if there exists $u_1, \ldots, u_n \in \Sigma^*$, $n \geq 1$ such that $\mathcal{I}_x(s(x)) = u_1$, $\mathcal{I}_x(t(x)) = u_n$ and $u_i \rightarrow_T u_{i+1}$, $1 \leq i < n$.*

This corollary defines a process of rewriting strings. Any string can be rewritten by replacing a substring which is any one side of an equation of the Thue system, with the string corresponding to the other side of the equation. The corollary (together with theorem 4.1) states that repeated rewriting in this manner is a sound and complete proof system for Thue systems.

There is still one problem remaining. The matching process is capable of matching only the most simple abstract data types and their algebraic specifications to Thue systems. Even the DEQUE abstract data type described earlier cannot be matched to a Thue system. This is because some of the operations have more than one parameter. For example, LEFT_PUSH has an additional parameter, albeit not of the abstract data type. Hence the matching methods need to be extended to a bigger subset. Such a matching method will be described, but first the extended subset is described below:

1. The constructors of the abstract data type (operations that return an abstract data type value) have to be unary with respect to the abstract data type. That is, they are permitted to have any number of parameters so long as only one of them is of the abstract data type. In addition, there can be selectors of any form.

2. There is exactly one constant of the abstract data type.

3. The algebraic specifications have to be of the form $t_1(x, \mathbf{y}) = t_2(x, \mathbf{z})$, where x is of the abstract data type, $\mathbf{y}$ and $\mathbf{z}$ are lists of values from types other than the abstract data type, and t_1 and t_2 consist of only constructors of the abstract data type. The algebraic specifications have to be universally quantified over all the parameters (i.e. $x, \mathbf{y}$ and $\mathbf{z}$).

Note that the DEQUE abstract data type example falls into this extended subset. By combining all the types other than the abstract data type into one composite type, a clean model of this extended subset can be achieved. This model is shown below. The composite

type of all the non-abstract data types is the auxiliary type in the generic package described in Section 4.2.

```
-- The auxiliary type definition
type AUX is ... ;

package P is

   -- The abstract data type definition
   type ADT is ... ;

   -- Constructors
   function C1(X:ADT;Y:AUX) return ADT;
   :

   -- Selectors
   function S1(X:ADT;Y:AUX) return AUX;
   :

-- Algebraic specification
--| axiom
--|     for all X:ADT;Y1,Y2,... :AUX =>
--|         Ci1(... Cim(X,Yjm)... ,Yj1) = Ck1(... Ckn(X,Yln)... ,Yl1)
--|           :

end P;
```

For this model, the matching process is defined in two steps. The first step involves converting the auxiliary type parameters to subscripts on the constructors. This step gets us back to the original subset though the number of operations may now be infinite. The result of applying this transformation on the above model is shown below:

```
-- The auxiliary type definition
type AUX is ... ;

package P is
```

```
        -- The abstract data type definition
        type ADT is ... ;

        -- Constructors
        function C_{1Y}(X:ADT) return ADT;
        :

        -- Selectors
        function S_1(X:ADT;Y:AUX) return AUX;
        :

    -- Algebraic specification
    --| axiom
    --|     for all X:ADT;Y_1,Y_2,... :AUX =>
    --|         C_{i_1 Y_{j_1}}(... C_{i_m Y_{j_m}}(X)... ) = C_{k_1 Y_{l_1}}(... C_{k_n Y_{l_n}}(X)... )
                :
                .

    end P;
```

Note that in the transformed model, the algebraic specifications have been converted to algebraic specification schemas. By performing the basic matching process now, a Thue system is obtained. The only difference is that the symbols in the resulting alphabet Σ may be subscripted by variables universally quantified over the auxiliary type. This difference however does *not* affect any of the theorems derived in this chapter, and hence the proof system can still be used in its original form[9].

Example 4.4: The DEQUE package of Example 1.2 is shown below after being rewritten to fit into the model of the subset. For simplicity, extra parameters of the auxiliary type are not added to operations that do not have this parameter to start with. The corresponding Thue system is also described:

```
package DEQUE_PACKAGE is
   type DEQUE is ... ;
   function CREATE return DEQUE;
   function LEFT_PUSH_E(D:DEQUE) return DEQUE;
   function LEFT_POP(D:DEQUE) return DEQUE;
```

[9] All proofs assume that strings are finite, but no assumption of the finiteness of the alphabet is made.

```
   function LEFT_TOP(D:DEQUE) return ELEMENT;
   function RIGHT_PUSH_E(D:DEQUE) return DEQUE;
   function RIGHT_POP(D:DEQUE) return DEQUE;
   function RIGHT_TOP(D:DEQUE) return ELEMENT;
--| axiom
--|    for all D:DEQUE;E:ELEMENT =>
--|       LEFT_POP(LEFT_PUSH_E(D)) = D,
--|       RIGHT_POP(RIGHT_PUSH_E(D)) = D,
--|       LEFT_POP(RIGHT_PUSH_E(D)) = RIGHT_PUSH_E(LEFT_POP(D)),
--|       RIGHT_POP(LEFT_PUSH_E(D)) = LEFT_PUSH_E(RIGHT_POP(D));
-- The other axioms are outside the subset and are therefore
-- ignored.
end DEQUE_PACKAGE;
```

The Thue system corresponding to the DEQUE package is the following: Σ is the alphabet $\{b_E, c, e_E, f\}$, for all E in ELEMENT. The b_E's correspond to the LEFT_PUSH$_E$'s, c corresponds to LEFT_POP, the e_E's correspond to the RIGHT_PUSH$_E$'s, and f corresponds to RIGHT_POP. The Thue system T is $\{\langle cb_E, \lambda\rangle, \langle fe_E, \lambda\rangle, \langle ce_E, e_E c\rangle, \langle fb_E, b_E f\rangle\}$.

The subsequent sections describe the theorem proving algorithm and its capabilities. In these subsequent sections, only mathematical notation will be used, the correspondence to Ada and Anna should be obvious from the discussion in this and other previous sections.

4.5 The Chromatic Theorem Prover

The theorem proving algorithm will be described in this section. This algorithm is based on term rewriting (corollary 4.10). The equations of the Thue system are used as rewrite rules during the theorem proving operations. However, a blind application of such rewrite rules can cause infinite loops during the theorem proving operations. To prevent such problems, the characters in the rewrite rules are colored in a manner described later. Infinite loops are avoided when the color assignments to the characters are also considered during the rewriting process. Any number of colors can be used with this algorithm. The more the colors, the more powerful (and slower) the algorithm. This algorithm has been named the *Chromatic Theorem Prover* (of order n, where n is the number of colors).

4.5.1 Some Terminology and Definitions

As before, let Σ be an alphabet (Σ^* is the set of all finite strings over Σ) and let T be a Thue system on Σ^*. Let $\mathcal{C}$ be a finite set of n colors with a total ordering relation $<$. $\mathcal{C}^*$ is the set of all finite sequences of colors from $\mathcal{C}$. Let $min(\mathcal{C})$ be the smallest color in $\mathcal{C}$. Define Γ to be $\Sigma \times \mathcal{C}$. That is, Γ is the set of all tuples $\langle a, b \rangle$ where $a \in \Sigma$ and $b \in \mathcal{C}$. Γ^* is the set of all finite strings over Γ. Intuitively, Γ^* is the set of all strings over Σ colored using the colors in $\mathcal{C}$.

A few functions are now defined on the above sets. π_1 is the *projection* from Γ and Γ^* to Σ and Σ^* respectively. Intuitively π_1 can be thought of as the operation of deleting the colors (or removing their significance) from the colored characters in Γ or the colored strings in Γ^*. π_2 is the *projection* from Γ and Γ^* to $\mathcal{C}$ and $\mathcal{C}^*$ respectively. Intuitively π_2 gives the colors of the colored characters in Γ or the color sequences of the colored strings in Γ^*. *minmax* maps $\mathcal{C}^*$ to $\mathcal{C}$. If c is in $\mathcal{C}^*$, then $minmax(c)$ is the smallest color which is larger than all the colors in c. Notice that *minmax* is undefined if c contains the largest color.

4.5.2 Constructing Rewrite Rules from Equations

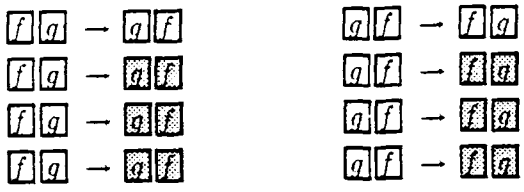
The first step in the algorithm is to convert the equations of T into rewrite rules (of order n). The rewrite rules are defined over Γ^*. The algorithm for constructing rewrite rules from equations is now given.

Let $\langle x, y \rangle$ be an equation in T. This equation translates into a set of rewrite rules which includes all (and nothing else) rewrite rules of the form $\alpha \rightarrow \beta$ ($\alpha, \beta \in \Gamma^*$) that satisfy the following conditions:

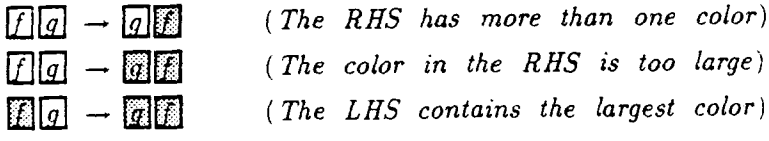
1. Either $\pi_1(\alpha) = x$, $\pi_1(\beta) = y$ or $\pi_1(\alpha) = y$, $\pi_1(\beta) = x$, and

2. $minmax(\pi_2(\alpha))$ is defined and all colors in the sequence $\pi_2(\beta)$ are the same and equal to $minmax(\pi_2(\alpha))$.

Example 4.5: Let $\Sigma = \{f, g\}$, and let $\mathcal{C} = \{$ *White*, *LightGray*, *Gray* $\}$, where *White* $<$ *LightGray* $<$ *Gray*. Then the rewrite rules generated from the equation $\langle fg, gf \rangle$ are:



Some examples of illegal rewrite rules are shown below:



Let R_T be the set of rewrite rules obtained in this manner from the equations of T. Note that if T and n are finite, then R_T is also finite.

4.5.3 The Neighbor Set

For every string x in Σ^*, a *neighbor set* N_x (of order n) is defined. It is however convenient to define a *colored neighbor set* $\mathcal{N}_x$ first. $\mathcal{N}_x$ is defined as follows:

1. $\chi \in \mathcal{N}_x$, where $\pi_1(\chi) = x$ and all the colors in $\pi_2(\chi)$ are the same and equal to $min(\mathcal{C})$.

2. If a, b, u, v are in Γ^*, $u \rightarrow v$ is in R_T and aub is in $\mathcal{N}_x$, then avb is also in $\mathcal{N}_x$.

3. Nothing else is in $\mathcal{N}_x$.

The neighbor set N_x can now be defined as follows: For every α in $\mathcal{N}_x$, $\pi_1(\alpha)$ is in N_x. Nothing else is in N_x.

Example 4.6: This is based on Example 4.5. Assume $x = ffg$. Then the following strings from Γ^* are in $\mathcal{N}_x$:

Hence the following strings from Σ^* are in N_x:

ffg, fgf and gff.

Theorem 4.11 *For any y, z in N_x, $T \vdash y = z$.*

Proof If y, z are in $\mathcal{N}_x$, then there must be some α, β in $\mathcal{N}_x$ such that $\pi_1(\alpha) = y$, $\pi_1(\beta) = z$, and both α and β have been derived by applying the rewrite rules of R_T repeatedly to χ, where $\pi_1(\chi) = x$ and all the colors in $\pi_2(\chi)$ are the same and equal to $min(C)$. Since this rewriting process is a restricted version of the proof system developed earlier, therefore $T \vdash x = y$ and $T \vdash x = z$. Hence $T \vdash y = z$. □

Theorem 4.12 *If Σ, T, C and x are finite, then $\mathcal{N}_x$ is also finite.*

Proof Let m be one more than the length of the longest string that occurs in any of the rewrite rules in R_T. Assign weights to the colors in C as follows: Assign 1 to the largest color. If w is the weight of any color, then the next smaller color is assigned a weight of mw (i.e., a geometric progression). Define the weight of any string α in Γ^* as the sum of the weights of the colors in the sequence $\pi_2(\alpha)$. Now consider any rewrite rule $\alpha \rightarrow \beta$ in R_T. By definition, the length of β is less than m, hence the weight of β is less than mw where w is the weight of $minmax(\pi_2(\alpha))$. The weight of α is at least mw since the colors in $\pi_2(\alpha)$ are all smaller than $minmax(\pi_2(\alpha))$. Therefore, in all the rewrite rules in R_T, the left-hand side has a larger weight than the right-hand side. Hence the application of a rewrite rule to any string results in a string with a smaller weight than that of the original string. Also, the smallest possible weight is 1. Assume that the weight of x is w_x. Hence starting from x, it is possible to perform rewriting at most w_x times before no more rewriting is possible. Also, if the length of x is l_x, then the largest string that can be generated is no more than $l_x + w_x(m-2)$ long[10]. Note that though it has been shown that any path in the rewriting process is bounded by w_x and hence finite, it has not yet been shown that the number of paths are finite. But since the length of any string generated by this rewriting process is bounded, there is also a bound on the number of ways in which one of these strings can be rewritten in one step. This in turn bounds the number of different paths that the rewriting process can take. Hence $\mathcal{N}_x$ is finite. □

Corollary 4.13 *It is possible to generate N_x from x in a finite amount of time given that Σ, T, C and x are finite.*

[10]The length of the right side of any rewrite rule can be at most $m-2$ longer than the length of the left side.

4.5.4 The Theorem Proving Step

The necessary background has now been established to explain how this algorithm attempts to prove $x = y$, for any $x, y \in \Sigma^*$. The algorithm evaluates the neighbor sets of x and y and then checks to see if these sets have any element in common. If they do, then the algorithm deduces $x = y$, otherwise it deduces $x \neq y$. More precisely, then algorithm deduces $x = y$ if $N_x \cap N_y \neq \phi$ (where ϕ is the empty set), otherwise it deduces $x \neq y$. The soundness of this algorithm follows from theorem 4.11.

4.5.5 Termination

For any $x, y \in \Sigma^*$, N_x and N_y can be evaluated in a finite amount of time (corollary 4.13). Also, since N_x and N_y are finite, the intersection of these sets can be evaluated in a finite amount of time. Hence, the Chromatic Theorem Prover terminates on all finite inputs.

4.6 Incremental Execution of the Algorithm

In this section, an algorithm that implements the Chromatic Theorem Prover is presented. This algorithm works in an incremental manner. The incrementality is obtained by noting certain properties of the application—runtime consistency checking. As mentioned in page 67, the problem in runtime consistency checking is to maintain a set Θ of all terms of the abstract data type whose values have *already* been evaluated by the program, and then attempt to deduce whether or not $t_1 = t_2$ is a theorem of the abstract data type for every t_1, t_2 in Θ. If so then the actual values of t_1 and t_2 must be equal to each other. A simplistic implementation of the algorithm would be to compute N_x for every new string x added to Θ, and then intersecting N_x with the neighbor sets of the strings already in Θ one by one. The time complexity of this implementation is 1 neighbor set construction and n intersections (where n is the number of strings already in Θ) every time a new string is added to Θ. This means that the time complexity increases as the program continues to run both because the size of Θ keeps growing and because the length of the strings typically increase. It is possible, however, to implement the algorithm incrementally in such a way that its time complexity does not depend on the size of Θ. The following observation is what makes it possible to have an incremental algorithm:

Lemma 4.14 *If $t \in \Theta$ and t is of the form fu, where f is in Σ and u is in Σ^*, then $u \in \Theta$.*

This is quite obvious for all it is saying is that for any term to be evaluated, all its subterms have to be evaluated first.

This algorithm maintains equivalence classes of terms rather than the terms themselves. Two terms are in the same equivalence class if they can be proved equal to each other based on the specifications. Whenever a new term is generated, the algorithm decides whether or not the term belongs to an already existing equivalence class. If so, then this term is added to the appropriate equivalence class; otherwise, a new equivalence class is created and this term is inserted into it. This process is depicted pictorially in Figure 4.1.

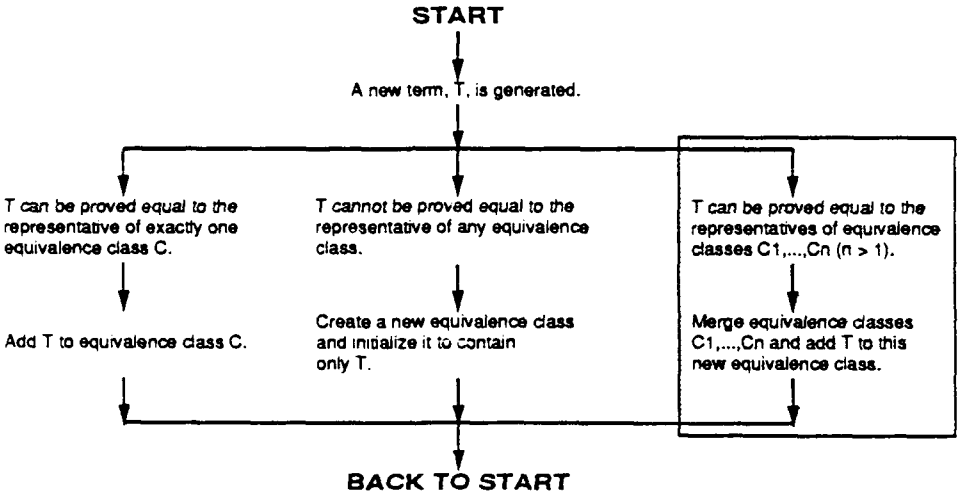


Figure 4.1: The Incremental Chromatic Theorem Prover

In this figure, only the high-level details are presented. The shaded third case in the figure arises due to the fact that there could exist multiple equivalence classes whose terms are equal to each other. That is, these equivalence classes should really have been one equivalence class. This happens because of the inability of the theorem prover to prove all possible theorems. Whenever case 3 is performed, this algorithm is *learning* in the process. In these respects, this algorithm is more powerful than the Chromatic Theorem Prover. However, the Chromatic Theorem Prover is a useful conceptual model of this algorithm for the purposes of analyzing the capabilities and limitations of this algorithm. The rest of this section will describe the algorithm in detail and then this algorithm will be shown to be at least as powerful as the Chromatic Theorem Prover.

4.6.1 Detailed Description of the Algorithm

The algorithm maintains a directed graph which it updates every time a new term is generated. The graph contains a special *initial* node which shall be referred to as n_i[11]. There is at least one path from n_i to each of the other nodes in the graph. The edges of the graph are labeled by tuples from Γ. Therefore, every path in the graph corresponds to a string from Γ^*. If P is a path, let $\gamma(P) \in \Gamma^*$ be the string corresponding to P. $\gamma(P)$ is defined recursively as follows: If P is just an edge (a path of length 1), then $\gamma(P)$ is the label on this edge. If P_1 is a path from n_1 to n_2 and P_2 is a path from n_2 to n_3, and if P_3 is the concatenation of the paths P_1 and P_2, then $\gamma(P_3)$ is $\gamma(P_2)\gamma(P_1)$. P is referred to as an *absolute path* if all colors in $\pi_2(\gamma(P))$ are the same and equal to $min(C)$. With every node n is associated a set $S(n)$ containing strings from Σ^*. A string x from Σ^* is in $S(n)$ if and only if there is a path P from n_i to n such that $\pi_1(\gamma(P)) = x$. As will be seen later, the $S(n)$'s correspond to the equivalence classes of Figure 4.1. It is also convenient to define another set $\mathbf{S}(n)$ containing strings from Γ^*. A string x from Γ^* is in $\mathbf{S}(n)$ if and only if there is a path P from n_i to n such that $\gamma(P) = x$.

The following four properties are the required invariants of this graph (except maybe while the graph is being updated):

1. If e_1, e_2 are edges originating from the same node, then $\pi_1(\gamma(e_1)) \neq \pi_1(\gamma(e_2))$.
2. If $x \in \Sigma^*$ corresponds to a term *already* generated by the program then there is an absolute path P originating from n_i to some node n such that $\pi_1(\gamma(P)) = x$. In this situation, the *value* of node n is said to be *defined*.
3. For every node n, if $x, y \in S(n)$ then $T \models x = y$.
4. If there is an absolute path P from n_i to n, and if $\pi_1(\gamma(P)) = x$, then $N_x \subset S(n)$.

These properties shall be referred to as the *graph invariants*. When the algorithm is initialized, the graph is initialized to contain just the node n_i and no edges[12]. Hence $S(n_i) = \{\lambda\}$. This initial graph trivially satisfies all the four graph invariants. Note that at the time of initialization, no terms have been generated by the program and therefore the second graph invariant holds.

[11] If there were more than one constant in the abstract data type, there will be an initial node corresponding to each constant. However, the graphs built from these nodes will always remain disjoint. Hence without loss of generality, the existence of only one constant can be assumed.

[12] The constant of the abstract data type is assumed to be evaluated when the program is initialized.

Whenever a new term is generated, the graph is updated. It will be shown that the update operation maintains the four graph invariants. Since the graph invariants hold on initialization, therefore (by induction) the graph invariants will hold after every update operation is performed. The update operation will now be described. Assume that the graph before the update is performed (G_0) satisfies the four above-mentioned properties. Let $x \in \Sigma^*$ correspond to the new term that has been generated. Let $x = ey$ where $e \in \Sigma$ and $y \in \Sigma^*$. Hence there is an absolute path P from n_i to some node n in G_0 such that $\pi_1(\gamma(E)) = y$. This follows from Lemma 4.14 and the fact that G_0 satisfies the second graph invariant. The update algorithm frequently performs the operation of *adding an edge* E to some node n_1 in the graph. This operation is described below:

1. Check if there is already an edge E' originating from node n_1 such that $\pi_1(\gamma(E)) = \pi_1(\gamma(E'))$.
2. E' exists:
   If E' has a greater color than E ($\pi_2(\gamma(E')) > \pi_2(\gamma(E))$, E' is replaced by E. Otherwise E is not added to the graph.
3. E' does not exist:
   Create a new node n_2 and insert E between n_1 and n_2.

Note that this operation of *adding an edge* has been designed to maintain the first and second graph invariants. In the algorithm below, all edges are added to the graph using this operation.

The update algorithm first adds an edge labeled $\langle e, min(C)\rangle$ to n. It now repeatedly performs the following sequence of operations until the graph cannot be updated any more:

1. Choose a path P in the current graph that includes at least one edge or *merge junction* (see below) created during the current updating operation such that there exists a rewrite rule in N_T of the form $\gamma(P) \rightarrow \beta$. Let P originate at node n_1 and terminate at node n_2.
2. Construct a path P' such that $\gamma(P') = \beta$ and add it to the current graph an edge at a time starting at n_1. Assume that this newly added path terminates at node n_3.
3. If the nodes n_2 and n_3 do not turn out to be the same, merge these nodes. This merging operation involves creating a new node n_4 and making all edges currently terminating in either n_2 or n_3 terminate at n_4. Make all edges originating from either n_2 or n_3 originate from n_4. However, if there are edges E_2 from n_2 to some

node n_2' and E_3 from n_3 to some node n_3' such that $\pi_1(\gamma(E_2)) = \pi_1(\gamma(E_3))$ then add only the edge with the smaller color and then perform the merge operation recursively on nodes n_2' and n_3'. Finally delete the nodes n_2 and n_3 from the graph. Note that the merge operation also maintains the first and second graph invariants.

Merge Junctions: Merge junctions are paths in the graph that are created as a result of the merge operation described above. To illustrate this, assume that nodes n_1 and n_2 in graph G_0 are merged, resulting in a modified graph G_1. If there were paths P_1 and P_2 in G_0 such that P_1 originated at n_1 and P_2 terminated at n_2, then the path P_2P_1 in G_1 is a merge junction created as a result of merging nodes n_1 and n_2.

Two examples are now shown to illustrate the working of this algorithm. In all the graphs of these examples solid, dashed and dotted edges correspond to the colors *White*, *LightGray* and *Gray* respectively.

Example 4.7: (based on Example 4.6) Three operations are performed—first $\mathcal{I}_x^{-1}(g)$ and then $\mathcal{I}_x^{-1}(f)$ twice—on the constant of the abstract data type, thus starting from the initial graph and modifying it three times. These four graphs are shown in Figure 4.2.

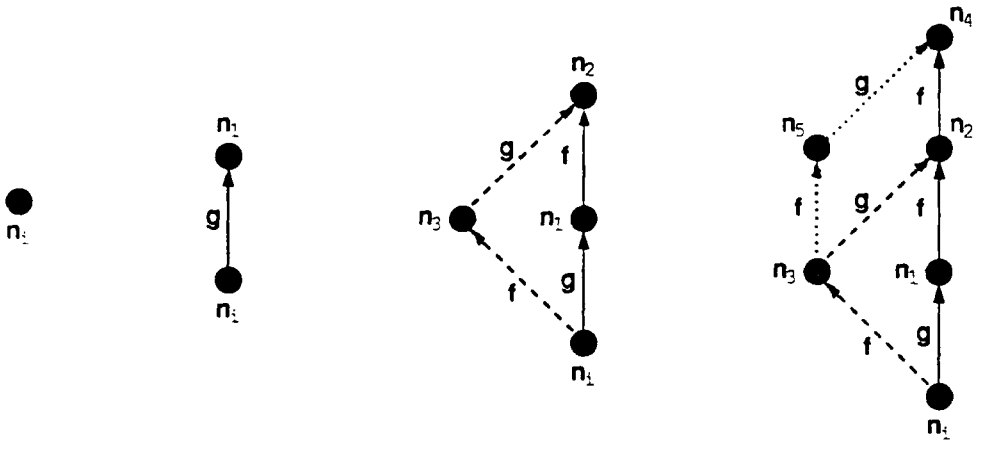


Figure 4.2: The Four Graphs of Example 4.7

In Figure 4.2, the first graph contains just the initial node. The second graph registers the fact that the operation $\mathcal{I}_x^{-1}(g)$ has been performed on the constant of the abstract data type. So far, no theorems have been proved. The third graph is the result of executing the next operation—$\mathcal{I}_x^{-1}(f)$. At this stage, the rewrite rules have been used to prove that

$\boxed{f}\boxed{g} = \boxed{g}\boxed{f} = \boxed{f}\boxed{g}$. However, when an attempt is made to add a path corresponding to $\boxed{f}\boxed{g}$ to the graph from n_i to n_2, it is seen that a path corresponding to $\boxed{f}\boxed{g}$ already exists between n_i and n_2. Hence, no new path is added. The fourth graph is the result of executing $\mathcal{I}_x^{-1}(f)$ one more time. The additional proofs derived at this point are exactly those of Example 4.6.

Example 4.8: Let $\Sigma = \{a, b, c, d, e\}$, $C = \{\boxed{White}, \boxed{LightGray}\}$ and $T = \{\langle a, b\rangle, \langle b, c\rangle, \langle c, d\rangle, \langle de, ed\rangle\}$. The following sequence of operations are performed one after the other on the constant of the abstract data type: $\mathcal{I}_x^{-1}(ea)$, $\mathcal{I}_x^{-1}(d)$ and $\mathcal{I}_x^{-1}(b)$. The four resulting graphs and the initial graph are shown in Figure 4.3.

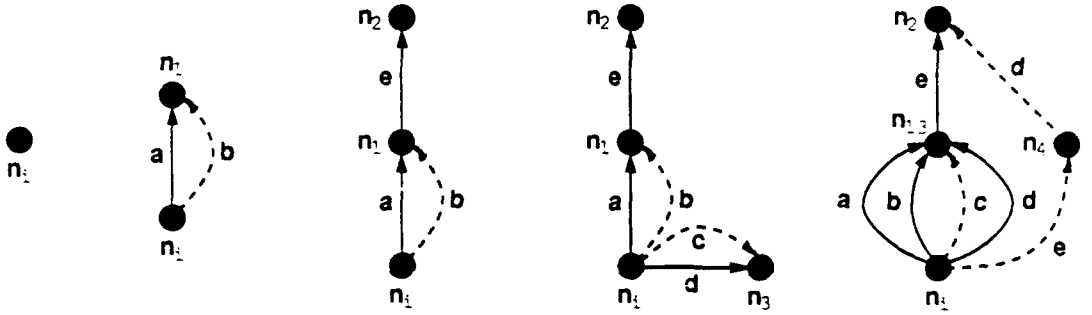


Figure 4.3: The Five Graphs of Example 4.8

The graphs of Figure 4.3 are similar to those of Figure 4.2. However, there is an interesting observation to be made in the fourth and fifth graphs. When the fourth graph is constructed, the fact that equivalence classes n_1 and n_3 are the same is not known. However, when $\mathcal{I}_x^{-1}(b)$ is executed, the gap in the proof is bridged. The result is a merging of the nodes n_1 and n_3 to form n_{13}. Hence this can be considered a learning process for the algorithm. The execution of $\mathcal{I}_x^{-1}9b)$ can be considered a *hint* to the algorithm. Also note that in the fifth graph, the path *de* from n_i to n_2 is a merge junction, since it was created as a result of merging nodes n_1 and n_3. Hence the need to extend the graph to include the path *ed* from n_i to n_2.

4.6.2 Comparison with the Chromatic Theorem Prover

On Page 88, four invariants of the of the graph used by the incremental algorithm were listed. These invariants (especially the fourth) are what relates the incremental algorithm

with the Chromatic Theorem Prover. In this section, informal proofs of the validity of each of these invariants is given. That the incremental algorithm is at least as powerful as the Chromatic Theorem Prover can then be concluded.

Theorem 4.15 (Invariant 1) *If e_1, e_2 are edges originating from the same node, then $\pi_1(\gamma(e_1)) \neq \pi_1(\gamma(e_2))$.*

Proof This holds trivially for the initial graph. The only way the graph gets updated is by the edge-adding operation and the merge operation. Both these operations have been designed to satisfy this invariant as has also been noted in the definitions of these operations. □

Theorem 4.16 (Invariant 2) *If $x \in \Sigma^*$ corresponds to a term already generated by the program then there is an absolute path P originating from n_i to some node n such that $\pi_1(\gamma(P)) = x$.*

Proof This can be shown by induction. The invariant holds trivially for the initial graph. Assume that this invariant holds for some intermediate graph G and a new term t is just generated by the program. Then, from Lemma 4.14, the term u, where $t = fu$ for some f in Σ, has already been generated. By the induction hypothesis, there is an absolute path P corresponding to u in G. The very first thing that the algorithm does is to add at the end of P an edge with label $\langle f, min(C)\rangle$. This creates the new path necessary to maintain the invariant. Also, the algorithm never replaces an edge colored by the smallest color. □

Theorem 4.17 (Invariant 3) *For every node n, if $x, y \in S(n)$ then $T \models x = y$.*

Proof This can also be shown by induction. The invariant holds trivially for the initial graph. Assume that the invariant holds for some intermediate graph G. G can be updated only by performing one of the following basic operations:

1. By adding an edge: This operation does not change the values of $S(n)$ of any existing nodes n. Assume that this operation creates a new node n_1 by adding an edge labeled $\langle f, c\rangle$ to some already existing node n_2. Then $S(n_1)$ contains all x in Σ^* of the form fy where y is in $S(n_2)$. Let x_1, x_2 be any two arbitrary elements of $S(n_1)$. Assume that $x_1 = fy_1$ and $x_2 = fy_2$. From the induction hypothesis, $T \models y_1 = y_2$. Therefore, $T \models x_1 = x_2$.

2. By merging two nodes: Assume that two nodes n_1 and n_2 are being merged to form n_{12}. This only happens when there exists a node n_3 such that there are paths P_1 from n_3 to n_1 and P_2 from n_3 to n_2 and either $\gamma(P_1) \rightarrow \gamma(P_2)$ or $\gamma(P_2) \rightarrow \gamma(P_1)$ is in R_T. Also assume that there is a path P_3 from n_i to n_3. Obviously, $\gamma(P_3P_1)$ in $S(n_1)$ is equal to $\gamma(P_3P_2)$ in $S(n_2)$. Therefore every element in $S(n_1)$ is equal to every element in $S(n_2)$ (from the induction hypothesis and the above observation). Hence for every x, y in $S(n_{12})$, $T \models x = y$. Now consider any arbitrary node n_4 such that $S(n_4)$ has changed as a result of this merge operation. The only way $S(n_4)$ can change is by new strings being added to it. These new strings will have to correspond to paths through n_{12}. Consider any such new path P_4P_5 where P_4 is a path from n_i to n_{12} and P_5 is a path from n_{12} to n_4. Without loss of generality, it can be assumed that P_4 ended at n_1 and P_5 originated at n_2 before the merge operation. Since $\gamma(P_4)$ is in $S(n_{12})$, therefore $\gamma(P_4)$ must be equal to $\gamma(P_3P_2)$ (proved earlier in this paragraph). Therefore, $\gamma(P_4P_5)$ must be equal to $\gamma(P_3P_2P_5)$. But $P_3P_2P_5$ was already in the graph before the merge operation—i.e. $\gamma(P_3P_2P_5)$ already existed in $S(n_4)$. Hence $\gamma(P_4P_5)$ has to be equal to all other existing strings in $S(n_4)$. □

Theorem 4.18 (Invariant 4) *If there is an absolute path P from n_i to n, and if $\pi_1(\gamma(P)) = x$, then $\mathcal{N}_x \subset S(n)$.*

Proof Define $\geq$, a binary relation over Γ^* as follows: $\alpha \geq \beta$ (where $\alpha, \beta \in \Gamma^*$) if and only if $\pi_1(\alpha) = \pi_1(beta)$, and for each color in $\pi_2(\alpha)$, the corresponding color in $\pi_2(\beta)$ is either the same color or a smaller color.

It will now be shown that if α is in $\mathcal{N}_x$ then there is a string β ($\in \Gamma^*$) in $S(n)$ such that $\alpha \geq \beta$. Note that it is possible for α and β to be the same string.

If α is in $\mathcal{N}_x$ then there are strings $\chi_1, \ldots, \chi_m$ in Γ^* such that $\pi_1(\chi_1) = x$, all colors in $\pi_2(\chi_1)$ are the same and equal to $min(\mathcal{C})$, $\chi_m = \alpha$, and for all $1 \leq i < m$, χ_{i+1} is derived from χ_i by the application of exactly one rewrite rule. Obviously, χ_1 is in $S(n)$. Assume that χ_i' is in $S(n)$, where $1 \leq i < m$, such that $\chi_i' \geq \chi_i$. Consider the rewrite rule used to derive χ_{i+1} from χ_i. Then there is another rewrite rule that differs from this one only in color which can be used to derive χ_{i+1}' from χ_i', where $\chi_{i+1}' \geq \chi_{i+1}$. In the graph, there must be a path corresponding to χ_i'. Assume that this path originates at n_1 and terminates at n_2. Note that this path must have been newly created at some time (either by adding an edge, or by merging two nodes). At this time, a path corresponding to χ_{i+1}' would have been

added between n_1 and n_2. In the process of adding this path, some already existing edges in the graph may have been encountered which may have been used as part of the path being added. The resulting path P will be such that $\gamma(P) = \chi''_{i+1}$, where $\chi''_{i+1} \geq \chi'_{i+1}$. Hence, $\chi''_{i+1} \geq \chi_{i+1}$. By induction, it is concluded that for all i, there exists a path corresponding to χ''_i in the graph such that $\chi''_i \geq \chi_i$. Hence, it can be concluded that β is in $\mathbf{S}(n)$.

Since $\pi_1(\beta) = \pi_1(\alpha)$, therefore, $\pi_1(\alpha)$ (which by definition is in N_x) is in $S(n)$. Hence, $N_x \subset S(n)$. □

Theorem 4.19 *The incremental algorithm just described is at least as powerful as the Chromatic Theorem Prover.*

Proof To realize this, all that has to be noted is that if an attempt is made to create two paths P_1 and P_2 originating from n_i such that $\pi_1(\gamma(P_1)) = \pi_1(\gamma(P_2))$, then this will result in the creation of *one* path P_3 such that $\gamma(P_1) \geq \gamma(P_3)$ and $\gamma(P_2) \geq \gamma(P_3)$. This is a consequence of invariant 1. Hence if the neighbor sets of two strings x and y have a non-empty intersection, and if there is an absolute path corresponding to x and y in the graph, then these two paths have to terminate at the same node. □

4.6.3 A Specialized Two-Color Algorithm

In this section, a specialized algorithm based on two colors is described. This algorithm is easier to implement that the general two color algorithm and has a better performance. Its capabilities[13] lie somewhere between those of the two-color and the three-color algorithms. This algorithm will be referred as the *order 2.5* algorithm. This algorithm is useful enough for most practical applications.

Assume that $C = \{c_1, c_2\}$ where $c_1 < c_2$. With two colors, each equation in T contributes exactly two rewrite rules to R_T. These rewrite rules correspond to the two directions of the equation. Terms on the left-hand side of the rewrite rules will all be of color c_1, while terms on the right-hand side will all be of color c_2. Therefore, in the discussion of this algorithm, only the equations need be referred to, their correspondence to the rewrite rules will be obvious.

As in the case of the algorithm described in Page 89, this update algorithm first adds an edge E labeled $\langle e, c_1 \rangle$ to n. It now repeatedly performs the following sequence of operations until the graph cannot be updated any more:

[13] *Capability* is defined precisely just before Theorem 4.23.

1. Choose a path P in the current graph that includes E such that there exists an equation in T of the form $\alpha\pi_1(\gamma(P)) = \beta$ for some α and β in Σ^*. Let P originate at node n_1 and terminate at node n_2.
2. Construct a path P' such that $\pi_1(\gamma(P')) = \beta$ and all the colors in $\pi_2(\gamma(P'))$ are the same and equal to c_2. Add this path to the current graph an edge at a time starting at n_1. Assume that this newly added path terminates at node n_3.
3. Construct a path P'' such that $\pi_1(\gamma(P'')) = \alpha$ and all the colors in $\pi_2(\gamma(P''))$ are the same and equal to c_2. Add this path to the current graph an edge at a time starting at n_2. Assume that this newly added path terminates at node n_4.
4. If the nodes n_3 and n_4 do not turn out to be the same, then merge these nodes as before.

There are two ways in which this algorithm is different from the two-color algorithm:

1. P only needs to correspond to a suffix of a rewrite rule's left-hand side. The rest of the left-hand side is constructed and added on to the graph. This is what happens when P'' is added to the graph. This difference has the consequence that merge junctions do not have to be considered during the update process. This is illustrated in Example 4.9 below. It can be shown that this difference does not increase the capabilities of the algorithm.
2. Only the edge E in P need be of color c_1. The other edges can be of any color. This increases the capability of the algorithm. This increase in capability is illustrated in Example 4.10.

Example 4.9: This is a repeat of Example 4.8, except that the order 2.5 algorithm is used instead of the order 2 algorithm. The four resulting graphs and the initial graph are shown in Figure 4.4.

Note that the fourth graph in this figure is different from the fourth graph in Figure 4.3. The nodes n_4 and n_5 are extra in this Figure as are the edges connecting them to the rest of the graph. This is due to the first of the above-mentioned two differences between the order 2.5 and the order 2 algorithm. As a result of this, when nodes n_1 and n_3 are merged, the nodes n_2 and n_5 are also merged, and the resulting graph is the same as the fifth graph in Figure 4.3. However, in Figure 4.3, a merge junction was considered, whereas this was unnecessary in this case.

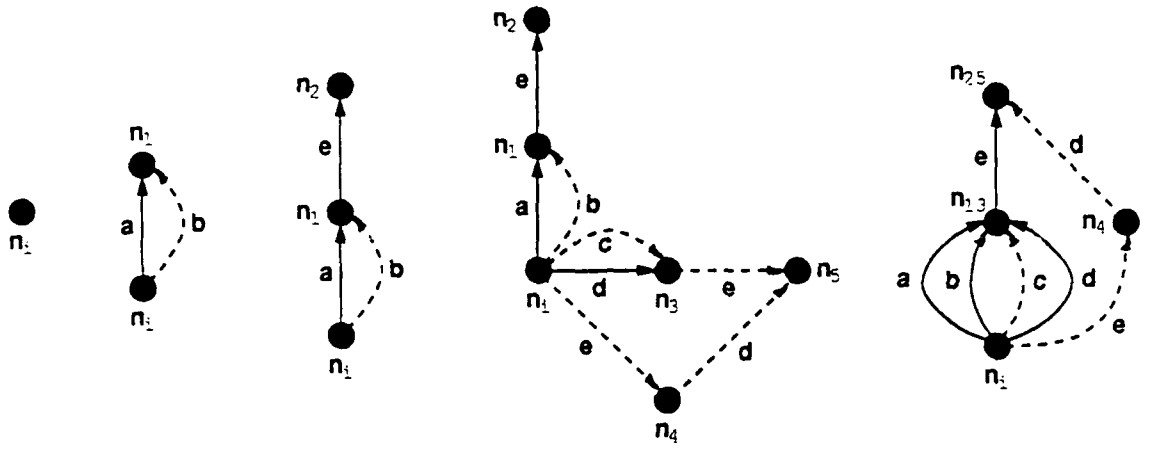


Figure 4.4: The Five Graphs of Example 4.9

Example 4.10: This example is based on the DEQUE abstract data type of Example 4.4. Assume that the sequence of operations, *dbca*, is evaluated (the subscripts are ignored for simplicity). The five graphs generated by the order 2 algorithm is shown in Figure 4.5, while the graphs generated by the order 2.5 algorithm is shown in Figure 4.6. Note that the order 2 algorithm could not deduce $dbca = \lambda$, but the order 2.5 algorithm could.

4.7 Capabilities of the Algorithm

Theorem 4.20 *Given sufficient hints, the incremental Chromatic Theorem Prover of any order (≥ 2) can prove any theorem.*

Proof Assume that $T \models x = y$ for some Thue system T and strings x, y in Σ^*. Then there is a sequence $z_1, \ldots, z_n$ of strings from Σ^* such that $z_1 = x$, $z_n = y$ and for $1 < i \leq n$, z_i can be obtained from z_{i-1} by the application of exactly one rewrite rule $\alpha \rightarrow \beta$, where $\alpha = \beta$ or $\beta = \alpha$ is an equation in T.

It is quite obvious that for $1 < i \leq n$, the Chromatic Theorem Prover of any order (≥ 2) can prove $z_{i-1} = z_i$. Now, if the operations corresponding to $z_1, \ldots, z_n$ are performed one after the other, the Chromatic Theorem Prover of any order (≥ 2) can prove $z_1 = z_2, z_2 = z_3, \ldots, z_{n-1} = z_n$. This means that in the final graph, the nodes corresponding to z_1 and z_2 are the same, the nodes corresponding to z_2 and z_3 are the same, and so on. So, the nodes corresponding to z_1 and z_n are also the same. Hence the prover is able to prove $x = y$ if

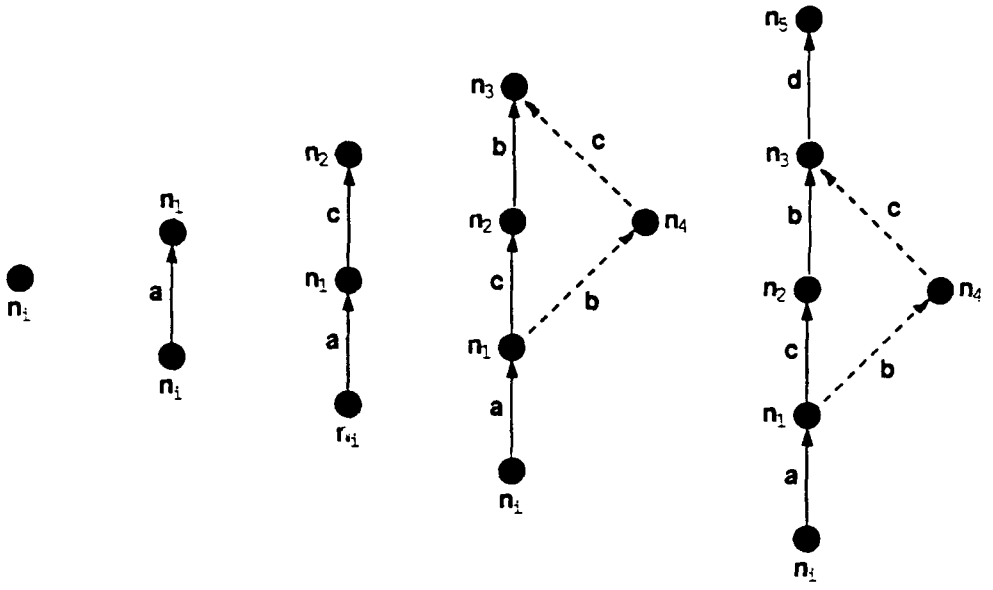


Figure 4.5: Order 2 Algorithm Graphs of Example 4.10

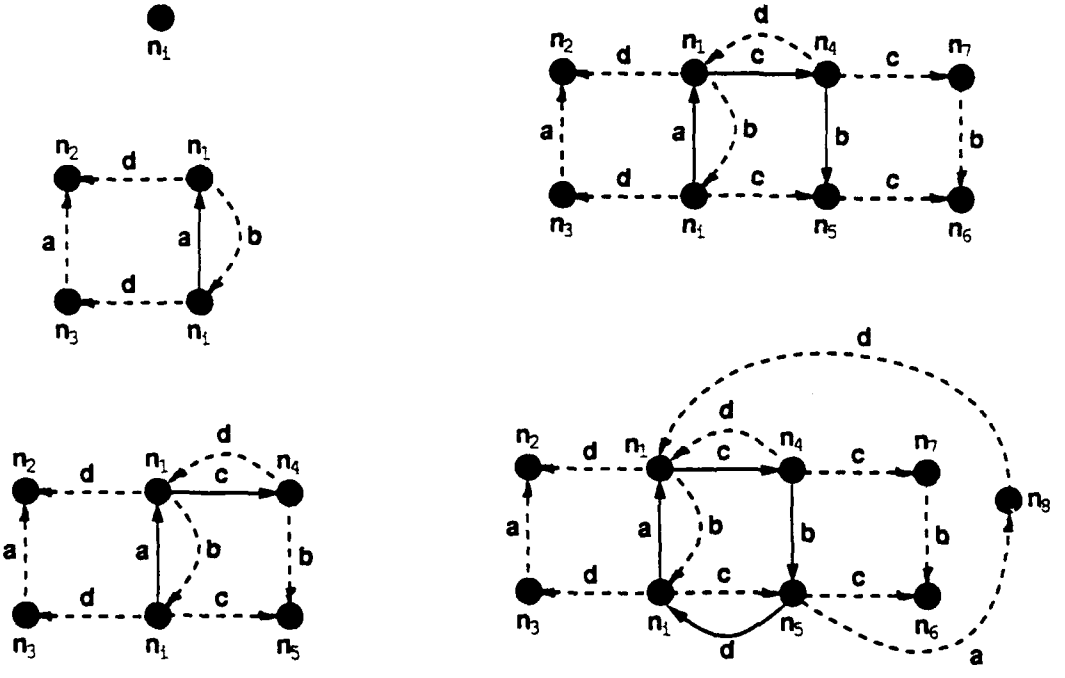


Figure 4.6: Order 2.5 Algorithm Graphs of Example 4.10

the operations corresponding to $z_1, \ldots, z_n$ are performed. Performing these operations can be considered giving hints to the prover. □

Theorem 4.21 *All theorems on stacks which are defined by the equation* POP(PUSH(S,X)) = S *can be proved by the Chromatic Theorem Prover of order 2 without any hints.*

Proof Let $\Sigma = \{f, g\}$, f corresponding to POP and g corresponding to PUSH. Then $T = \{\langle fg, \lambda \rangle\}$. Let $C = \{$ *White* *LightGray* $\}$. The rewrite rules are:

$$\begin{array}{rl} \boxed{f}\boxed{g} & \rightarrow \lambda \\ \lambda & \rightarrow \boxed{f}\boxed{g} \end{array}$$

However, note that the theorem proving algorithm is such that the second rewrite rule will never be used. Also, since the application of the first rewrite rule does not introduce a new color, the algorithm is capable of deriving all terms from a term α that can be derived from α by a repeated application of the first rewrite rule. Since only one color will be present in all terms, colors are omitted from the following discussion for convenience. It follows that if the theorem prover finds a proof for $\alpha = \beta$, where α and β are in Σ^*, then the proof has to be of the form $x_1, \ldots, x_m, \ldots, x_n$ where $x_1 = \alpha$, $x_n = \beta$, for $1 < i \leq m$, x_i is obtained from x_{i-1} by the application of the first rewrite rule (without considering the colors) and finally, for $m \leq i < n$, x_i is obtained from x_{i+1} by the application of the first rewrite rule (again without considering the colors).

However, all proofs for $\alpha = \beta$ need not be of this form. But it is sufficient to prove that if there is a proof, then there is at least one proof of the above form. Assume that there is a proof for $\alpha = \beta$. Then there is a sequence $y_1, \ldots, y_p$ such that $y_1 = \alpha$, $y_p = \beta$ and for $1 < i \leq p$, y_i is obtained from y_{i-1} by the application of either one of the above rewrite rules (without considering the colors). A transformation process will now be described which will convert this proof into a proof of the form generated by the Chromatic Theorem Prover.

Assume that the sequence $y_1, \ldots, y_p$ is not of the above form. Then there has to be a subsequence in $y_1, \ldots, y_p$ of the form:

$y_i \rightarrow y_{i+1} \rightarrow y_{i+2}$

y_{i+1} is derived from y_i by exactly one application of the rule $\lambda \rightarrow fg$, and y_{i+2} is derived from y_{i+1} by exactly one application of the rule $fg \rightarrow \lambda$. There are two cases to consider:

1. $y_i = \alpha\beta$, $y_{i+1} = \alpha fg\beta$, $y_{i+2} = \alpha\beta$: In this case, this subsequence is redundant for

the purpose of the proof, and therefore, y_{i+1} and y_{i+2} can be thrown away and the remaining sequence still represents a valid proof.

2. Either $y_i = \alpha\beta fg\gamma$, $y_{i+1} = \alpha fg\beta fg\gamma$, $y_{i+2} = \alpha fg\beta\gamma$; or $y_i = \alpha fg\beta\gamma$, $y_{i+1} = \alpha fg\beta fg\gamma$, $y_{i+2} = \alpha\beta fg\gamma$: In either situation there is an alternate subsequence in which y_i and y_{i+2} remain the same, but $y_{i+1} = \alpha\beta\gamma$. This new subsequence still represents a proof, but differs from the original subsequence in that the order in which the two different rewrite rules are applied is reversed.

Repeated application the above transformations until they cannot be applied any more obviously results in a sequence of the required form. Now all that is left is to show that this process will in fact terminate. For each sequence, define a weight that is the sum of the distances of each application of the rules $fg \rightarrow \lambda$ from the left-hand side and the distances of each application of the rules $\lambda \rightarrow fg$ from the right-hand side. It is quite obvious that each application of the above transformation decreases the weight of the sequence. The weight of a sequence cannot be less than 0, and hence the transformation process will eventually have to terminate. □

Theorem 4.22 *For any finite n, there are Thue systems in which certain theorems cannot be proved with the order n algorithm.*

Proof Let $\Sigma = \{a_1, \ldots, a_{2n}\}$, and let $T = \{\langle a_1, a_2\rangle, \langle a_2, a_3\rangle, \ldots, \langle a_{2n-1}, a_{2n}\rangle\}$.

The algorithm of order n cannot prove $a_1 = a_{2n}$ without any hints. For the only way to prove this is to perform the following rewrite operations on a_1 and a_n (the colors in the symbols below are left out for simplicity):

$a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_i$

$a_{2n} \rightarrow a_{2n-1} \rightarrow \ldots \rightarrow a_i$

Since there are a total of $2n + 1$ symbols in the two rewriting operations above (since a_i occurs twice), therefore the longer of the two sequences has to contain at least $n+1$ symbols. Also each symbol in the sequence has to have a color smaller than the color of the symbol to its left. But there are only n colors. Hence it is impossible to perform the above rewrite operations. □

If $T \models \alpha = \beta$, then the Chromatic Theorem Prover has the *capability* to prove this theorem if and only if in any graph that contains absolute paths corresponding to α and β, these paths terminate at the same node.

One algorithm is considered to have a *greater capability* than another algorithm if the former has the capability to prove all the theorems that the latter has the capability to prove and in addition has the capability to prove some more theorems.

Theorem 4.23 *The order 3 algorithm has a greater capability than the order 2.5 algorithm, which in turn has a greater capability than the order 2 algorithm.*

Proof Example 4.10 has already demonstrated that there are situations in which the order 2.5 algorithm has more capabilities than the order 2 algorithm. Also, as an obvious consequence of the algorithm definitions, the order 2.5 algorithm does everything the order 2 algorithm does and more. Hence the order 2.5 algorithm has a greated capability than the order 2 algorithm.

In Example 4.9, if the order 3 algorithm was used instead of the order 2.5 algorithm, then $a = d$ could have been proved without having to evaluate the operation b. Hence there are situations in which the order 3 algorithm has more capabilities than the order 2.5 algorithm. To conclude this proof, it has to be shown that the order 3 algorithm is capable of proving all theorems that the order 2.5 algorithm is capable of proving. This is shown informally below.

The order 2.5 algorithm can be redefined using three colors instead of two. Whenever P'' (Page 95) is not empty, then make all colors in P' (defined one paragraph earlier in Page 95) the same and equal to the third color. This redefinition helps compare the order 2.5 and the order 3 algorithms more easily. It is quite obvious now that the rewrite rules used by the order 2.5 algorithm is a subset of those used by the order 3 algorithm. The order 2.5 algorithm does not include rewrite rules whose left-hand sides do not contain at least one symbol of the smallest color. Hence, when these two algorithms perform without hints, the order 3 algorithm does better since it has more rewrite rules available. Even though the order 2.5 algorithm does perform a certain form of *lookahead* operation, it is still constrained by the rewrite rules available to it. □

4.8 Miscellaneous Topics

4.8.1 Going Outside the Subset

The algorithms described in this chapter work on a limited subset of all possible abstract data types. Even though a large number of interesting real-life application fall into this

subset, there are times when it is necessary to go out beyond the subset. In this case, the algorithms of this chapter will not work as is.

A common situation is one in which there are non-unary operations. A typical example is of a *set* package. This package might define operations like *union* and *intersection* which are binary. However, these operations can be defined in terms of other unary operations like *add* and *remove* using Anna subprogram annotations. Once this is done, the binary operations no longer have to be specified axiomatically. Hence, one solution to the problem is to split the abstract data type operations into those that fall into the subset and those that do not, and then to define the latter operations in terms of the former operations using Anna subprogram annotations. A clean methodology however has to be worked out to interface the two different modes of checking dealt with in this thesis. For example, how does the graph maintained by the theorem prover get updated as a result of executing a binary operation defined in terms of other unary operations? This is a topic for future research.

The above solution will not work always. In such situations (for example, a *complex number* package), the incremental methodology can still be used, but a more powerful theorem prover will be necessary. This theorem prover must be capable of performing in an incremental fashion. The Knuth-Bendix theorem prover is one that can be used to replace the theorem prover described in this chapter. It is not useful to replace the Chromatic Theorem Prover by the Knuth-Bendix theorem prover when the abstract data type is within the subset requirements of the former, since the former prover is much faster.

4.8.2 Undefinedness of Expressions

In Anna, undefined expressions are not considered in ascertaining the correctness of the program[14]. Hence, it is possible in Anna to trivially satisfy all algebraic specifications by implementing all operations as infinite loops, or such that they always terminate abnormally by raising an exception.

The Chromatic Theorem Prover cannot recognize when an operation is undefined and when it is not. The prover could come to wrong conclusions as a result of this. In Example 4.8, if the operation c was undefined, then it would be wrong to merge the nodes n_1 and n_3. One way to solve this problem is to actually evaluate all operations corresponding to two paths being merged. This may sometimes be inefficient. A clean way to do this is a

[14] Partial semantics.

topic for future research.

4.8.3 Concurrent Algebraic Specification Checking

It is quite easy to set up the theorem prover on a separate processor since it does not share data with the underlying program (unlike in the case of generalized assertion checking). The only communication required is through the subprogram calls defined in the generic theorem prover package specification. There is not much of an overhead in sending the necessary messages between the processors since the messages are quite compact.

As in the case of generalized assertion checking, there is the problem of the underlying program running inconsistently while earlier check requests are still being processed by the checking system. It may be acceptable to let the underlying program execute, but wait every time it invokes an abstract data type operation until any earlier checks are completed.

Chapter 5

Debugging Formally Specified Programs

Specification languages present an opportunity to develop new techniques in all phases of software production. There are two aspects to developing these new techniques—*developing a new capability*, and *defining methods of applying this capability*. The previous chapters of this thesis have described a new capability—automatic runtime consistency checking of programs against their formal specification. In this chapter, a method of applying this capability, namely *debugging based on formal specifications*, is described[1].

Debugging has always been a creative endeavor. A great deal of ingenuity and effort has gone into using the current generation of debugging tools to discover errors in programs. The main point of this chapter is that much more powerful debugging techniques can be developed based on formal specifications and the existence of a runtime consistency checking tool. These new techniques utilize both the high-level concepts used in specifications and the abstraction and information hiding constructs used in modern programming languages.

5.1 The Anna Debugger

A basic understanding of the *Anna Debugger* is essential to appreciate the debugging techniques described in this chapter. A detailed description is available in Appendix C.

The *Anna Transformer* transforms Anna specifications into Ada checking code. This

[1]The methodologies described in this chapter are a result of debugging experiments performed along with Shuzo Takahashi and David Luckham.

checking code reports an error whenever it detects that the underlying program has become inconsistent with the original specifications. The inconsistencies are reported through the Anna Debugger. The Anna Debugger specifies both where the inconsistency took place as well as the particular Anna specification that the program became inconsistent with.

Control is transferred to the debugger before the program starts running and then every time an inconsistency is detected. Through the debugger, the programmer can suppress the checking of annotations, or un-suppress the checking of annotations which were suppressed earlier. Consistency checking with respect to suppressed annotations is not performed. Suppressing annotations not only makes the consistency checking process more efficient (by not performing irrelevant checks); it also allows the programmer to concentrate on particular portions of the program being debugged. The Anna Debugger provides a few more capabilities, like controlling when the exception ANNA_ERROR is raised, but these capabilities are not made use of in the techniques described in this chapter.

A schematic diagram of how the Anna Debugger interacts with the generalized assertion checking subsystem is shown in Figure 5.1 and the corresponding diagram for the algebraic specification checking subsystem is shown in Figure 5.2. These figures expand on Figure 1.1 and give more details of the interaction with the Anna Debugger. The Anna Debugger provides a window based user-interface. Figure 5.3 illustrates the screen layout when an inconsistency is detected. This figure (and other similar figures in this chapter) does not go into the full details of the debugger user-interface; rather it concentrates on those aspects relevant to the debugging methodology presented here.

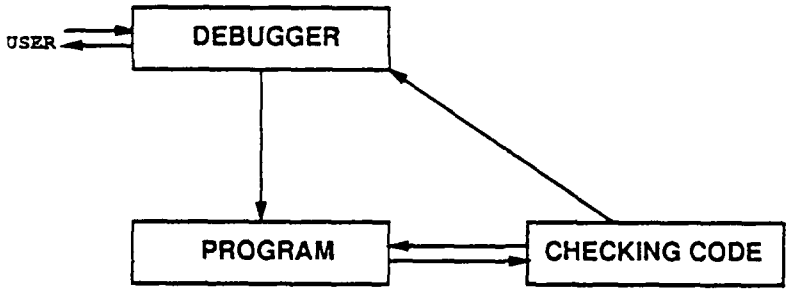


Figure 5.1: The Generalized Assertion Checking/Debugging Subsystem

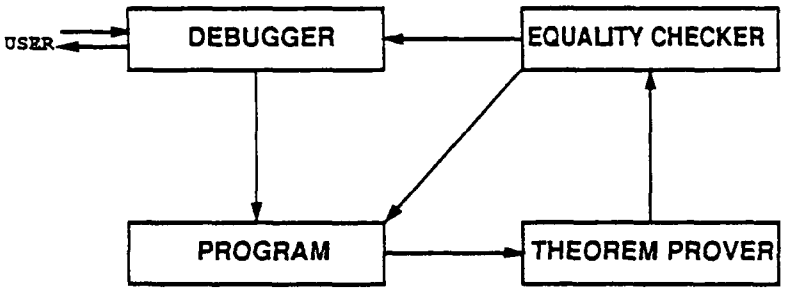


Figure 5.2: The Algebraic Specification Checking/Debugging Subsystem

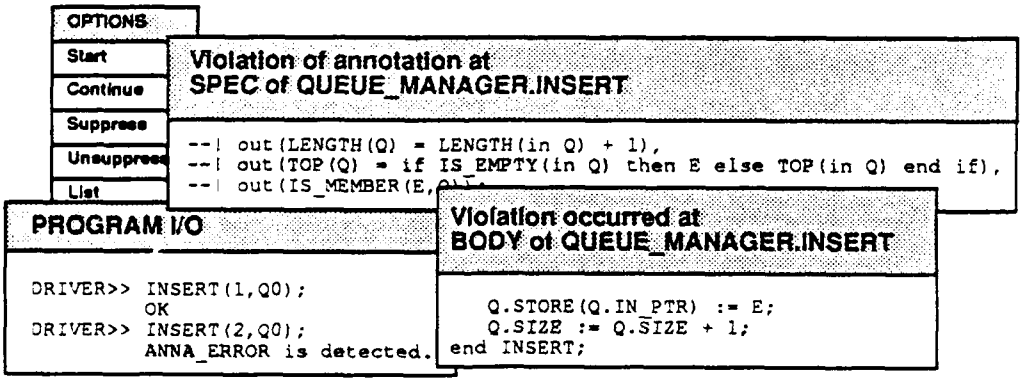


Figure 5.3: A Typical Anna Debugger Screen Layout

5.2 Operation Sequences and Structural Levels

The execution of a sequential program can be modeled as the performance of a *sequence* of *operations* in a particular order. Each of these operates on the program data. Figure 5.4 illustrates this pictorially. The oval denotes the program data, while the rectangles denote the operations. The solid arrows show the execution sequence while the dotted arrows signify access of program data by the operations.

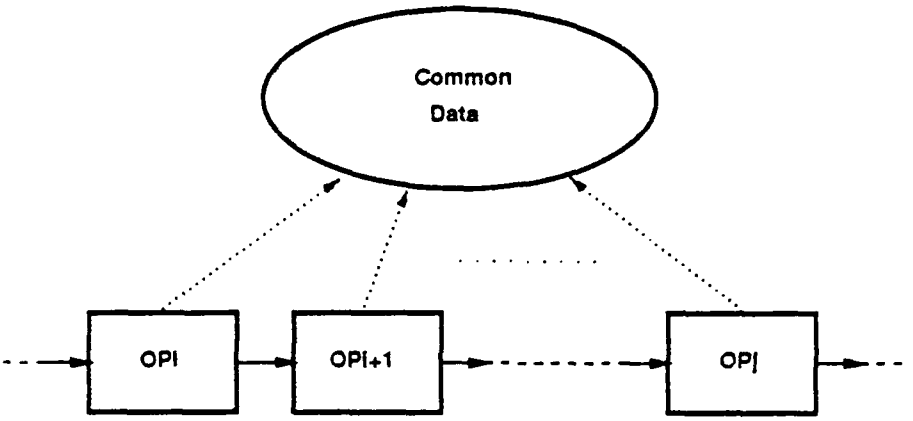


Figure 5.4: Operation Sequences

The operations and data of a program can be abstracted at different *levels* of the program. This abstraction process gives the program its structure. Some of these structural levels of abstraction are discussed below:

The Operating System Command Language Level: The operations at this level are the programs themselves. Hence, this is not really a structural level of a program, but a higher level where programs themselves form the basic operations. For example, a *UNIX shell script* is made up of programs like *awk*, *grep*, etc. The data at this level is the program input and output, for example, disk files.

The Program Level: Programs typically comprise of modules. Modules export functions and procedures that can be called from other modules. These functions and procedures are the visible module operations and form the sequence of operations at the program level. The data at this level are the variables and data structures declared within the modules.

These form the *states* of the modules. In Ada, modules are implemented as packages.

The Module Level: At the module level, only one module is considered. The execution of an operation of this module is influenced by the execution of earlier operations of the same module and operations of the modules that this module depends on. The sequence of operations at the module level is therefore the subsequence of the program level operations whose operations are either in this module or dependent modules. The data at this level is the states of these modules.

The Module Operation Level: The operations at this level are the sequence of statements within the module operation. The data at this level is the set of all program variables and data structures defined within the module operation.

The Compound Statement Level: This level is quite similar to the module operation level, the operations are the statements within the compound statement, and the data is the set of all program variables and data structures defined within the enclosing module operation.

The Machine Instruction Level: The operations at this level are the machine instructions. The data at this level are the machine registers and memory. As in the case of the operating system command language level, this level is also not considered to be a program level.

Summary: There are a few points to be noted based on what has been described above:

1. At all levels, there is a sequence of operations which interact by sharing some common program data.

2. The program data accessible to an operation at any particular level includes the data accessible at higher levels.

3. The choice of a sequence is more flexible at some levels that others. For example, at the module level, a test program can be written to invoke the module operations in any chosen order, while at the module operation level, the sequences of statements are in some respects hard-wired.

4. An operation at any particular level is in general composed of other operations which can be from any level.

5.3 Assumptions about Specifications

It is assumed that relevant aspects of the program behavior are explicitly written out as formal specifications. These specifications are assumed to occur at all structural levels of the program. The concepts[2] used in these specifications are defined at the appropriate level. For example, at the module level, where the details of the implementation of the abstract data type are not available, the concepts are high-level attributes of the abstract data type (independent of the implementation). The writing of the specifications is assumed to take place at some earlier stage in the software development with the aid of other specialized tools like specification analyzers[3]. Hence, for the purpose of the debugging methodology described here, the specifications are assumed to be correct. The implementation of the concepts used in these specifications is also assumed to be correct. These concepts are assumed to have been debugged earlier, possibly during their development. Though the specifications are assumed to be correct, they may not be complete. Hence, there may be certain aspects of the program behavior that may not have been specified.

The assumptions about specifications mentioned above are necessary for the purpose of developing the debugging methodology described later in this chapter. In real-life situations, though these assumptions may not strictly apply always, they will still be valid to some extent. The methodology described here will still aid the programmer in these situations.

5.4 Two-Dimensional Pinpointing

The debugging methodology described in this chapter is termed *two-dimensional pinpointing* [76]. It refers to the process that starts at the detection of the first inconsistency and results in the narrowing down of the location of the problem.

In general, two strategies are involved in debugging a program. First, tests involve executing sequences of program operations and using the Anna Consistency Checking System to compare their runtime behavior with the specifications. Second, only the highest level

[2]Concepts are functions used in specifications.

[3]The Anna Consistency Checking System is not designed for writing specifications, rather it assumes that the program and the specifications are already available.

specifications are checked unless a violation occurs. Thus, the debugging problem is regarded as having two dimensions: *the length of the test sequence*, and *the structural levels of the program*.

Two-dimensional pinpointing starts if and when inconsistencies arise. When an inconsistency arises, a *region of suspicion* is defined. It is a region of the program text (including specifications), where the problem that caused the inconsistency is guaranteed to be present. It may include not only the (sub) unit whose execution propagated the inconsistency, but also other units at the same level that are related to it, or preceded it in the test sequence. The goal of two-dimensional pinpointing is to reduce the region of suspicion as much as possible.

Pinpointing strategies utilize the hierarchical structure of the program. New, more detailed specifications about program behavior, are added to the previous specifications—this is called *augmenting* specifications. New specifications are first added at the same level as the one that was violated. They should be related to the violated one in such a way that they are likely to be violated by the same test. The same test sequence is then repeated. A violation of a new specification may reduce the region of suspicion by providing more information, and possibly by occurring earlier in the test sequence. At any one time only a few high level specifications are checked on a given test sequence.

When the region of suspicion has been reduced as much as possible by augmenting specifications at one level, the next lower level is considered. At this new level, the checking of specifications of those nested units that remain in the region of suspicion is activated. This strategy is repeated at progressively lower levels of the program. Each time a violation occurs, the length of the test sequence is shorter, or the level of the violated specification is lower—i.e., one of the two dimensions of the test is reduced. Once the region of suspicion has been reduced sufficiently, an attempt can be made to *repair* the program fragment corresponding to this region.

This hierarchical use of specifications may be employed with a range of informal or formal methods. For example, in pinpointing, the methods whereby new specifications are created may be highly intuitive, such as "guess-and-test". Alternatively, new specifications may be formally proved to imply the violated ones before they are tested. When repairs are made, goal-oriented techniques utilizing the specifications can be used [34]. Or repairs can be proved consistent with specifications that were previously violated.

The advantages of debugging with specifications over present debugging methods and

tools include:

- *The debugging problem is precisely defined.*
  The set of specifications to be tested constitute formal definition of the behavior to be tested. If, later on, new specifications are required, analysis of their relationship to the old specifications will indicate what further tests need to be performed.

- *Violations of specifications are detected automatically.*
  The task of searching output traces in order to recognize errors is eliminated.

- *Violations can be analyzed for their effect on users of the software.*
  Violation of a high level specification in the interface of a module or package indicates explicitly which facilities are unreliable. Users of other facilities may be unaffected.

- *During debugging, new specifications can be expressed formally at any program level, and then tested by the same methods and tools.*
  A user interacts with the Anna Consistency Checking System by formulating new specifications and submitting them to tests. The user no longer has to deduce whether an abstract property of the program is violated from more primitive data.

- *Very complex tests can be formulated easily and checked automatically.*
  For example, specifications against side-effects on global data are easily formalized and tested. This is difficult to do with standard debuggers.

- *Such methods and tools are independent of the language implementation.*
  For example, the Anna Consistency Checking System can be used with any Ada compiler and runtime implementation.

- *These methods apply equally well to concurrent programs.*
  For concurrent software, the methods use specification languages that extend Anna by providing new constructs for concurrent behavior—e.g., TSL [72].

The two-dimensional pinpointing methodology is now summarized as a set of three general guidelines for applying this methodology. The example in Section 5.6 illustrates the application of these guidelines during the process of debugging a QUEUE package. Before going into these guidelines, certain ramifications of the assumptions made about specifications in Section 5.3 are listed:

- *No initial (starting) specification may be changed.*
  This follows from the assumption that the specifications are correct.

- *New specifications may be added provided they are consistent with the old ones.*
  This follows from the assumption that the specifications, though correct, may still be incomplete.

- *Repair may involve any change to executable code, but data structures may be changed only to the extent that they are not constrained by the specifications.*
  This also follows from the assumption that the specifications are correct.

- *An inconsistency of the program behavior with an intuitive intention may occur.*
  In this case, the intuitive intention has not been formally expressed, which may indeed be the case since the specifications are assumed to be incomplete.

- *The operation in which the inconsistency is detected by the Anna Consistency Checking System is not necessarily at fault. The earlier operations in the sequence have to be included in the region of suspicion.*
  This follows from the assumption that specifications may be incomplete. Hence, when the operation in which the inconsistency was detected began execution, the common data (see Figure 5.4), may have already been inconsistent due to a problem in one of the earlier operations in the sequence. This was not detected since this kind of inconsistency may not have been formally specified against explicitly.

The guidelines are now given below:

1. Adding specifications at the current level: An attempt is made to add new specifications at the level being tested. This choice is indicated whenever there seem to be missing specifications. It is obviously necessary whenever the program behavior is inconsistent with an intuitive intention although no specification was violated. This strategy usually results in reducing the region of suspicion by shortening the test sequence in which a violation occurs.

2. Going down to a lower level: The program is tested at the next lower level. The specifications at this level are activated. The region of suspicion will be reduced if one of these newly activated specifications is violated. The original level will then be excluded

from the new region of suspicion. This choice is taken when there does not seem to be any *missing specifications* at the earlier level.

3. Confidence in the completeness of specifications: If there is confidence in the completeness of specifications that constrain a certain intermediate point during the execution of the sequence of operations, and this set of specifications is not violated, then the operations in the sequence that occur before this point can be excluded from the region of suspicion.

The rest of this chapter illustrates the application of two-dimensional pinpointing to debugging Ada packages. The structure of the Ada package is described followed by a detailed example where the debugging methodology is carried out on a QUEUE package. Many topics, such as the use of algebraic specifications, are omitted. A fuller treatment is given in [75]. Techniques are described there to construct specifications so as to justify the postulate that debugging starts with *correct* specifications. More detailed examples of pinpointing are given, and the reasoning to justify various steps is discussed in detail, some of it formally.

5.5 Ada Packages

Ada packages are complex units of abstraction with several structural levels. While the package executes, it performs a sequence of package operations. Details of the structural levels of packages are given below. To debug a package, a test sequence—a sequence of package operations—is first chosen. The package is executed on these operations one after the other. If an inconsistency is detected, the process of two-dimensional pinpointing is commenced. At this point, only the specifications at the highest structural level (the package visible level) are activated.

An Ada package can be considered to be a composition of entities at different *structural levels*. A simple package may have the following structural levels:

- *The visible level:*
  The visible level of the package is where the interface of the package to the rest of the program is defined. The visible level typically consists of a *private type* and *subprogram* declarations.

- *The data level:*

At this level is the definition of the data structures and the data objects of the package. The *private part* of the package is also part of the data level.

- *The subprogram level:*
  This level consists of the actual bodies of the subprograms inside the package body.

- *The statement level:*
  This level consists of the sequence of statements within the subprogram. Actually, this level can be split up into more than one level. For example, a compound statement (e.g. a loop) may be considered at a *higher* structural level than the simple statements within it.

In the example of Section 5.6, a QUEUE package with the standard operations of INSERT, REMOVE, etc. is presented. The structural levels of this package is illustrated in Figure 5.5.

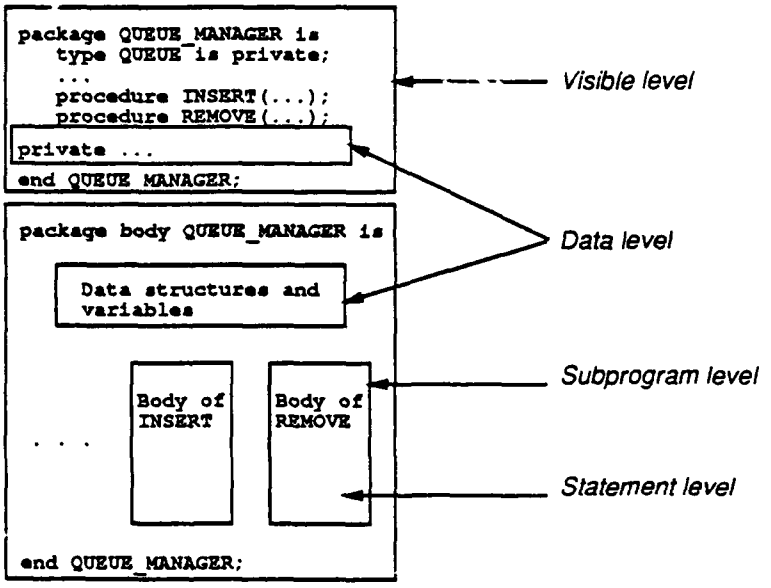


Figure 5.5: Structural Levels within the QUEUE Package

The test sequences consist of visible package operations. In the example of Section 5.6, the test sequences consists of repeated applications of the operations INSERT and REMOVE.

5.6 An Illustrative Debugging Session

In this section, the QUEUE_MANAGER package written in Ada/Anna is tested and debugged, using the Anna Consistency Checking System. The example, although simple for brevity, illustrates the application of the two-dimensional pinpointing methodology.

5.6.1 The QUEUE_MANAGER Package

The QUEUE_MANAGER package provides an abstract queue type, procedures, functions, and exceptions. The Ada declaration of the package contains the abstract Anna specification which is visible to the users of the package. This Ada/Anna package visible specification defines the behavior of the facilities (types and operations) provided by the package.

The Ada package body, including the Ada private part, contains an implementation, the details of which are hidden from the users of the package. This hidden part, which has three structural levels, also contains local annotations specifying how the implementation works. The hidden annotations refer to the hidden implementation details, and thus will be different for different implementations. The stages during the development of the QUEUE_MANAGER package are shown below:

1. Types and exceptions are declared. In particular, the abstract data type QUEUE is declared as an Ada private type.

2. The functions IS_MEMBER, LENGTH, and TOP are declared. These are referred to as *basic concepts* because they are used to specify all of the other QUEUE_MANAGER operations.

3. The rest of the operations are declared and specified in terms of the concepts. Actually, two of the other package operations, IS_EMPTY and IS_FULL, which have very simple specifications in terms of the basic concepts, are also used as concepts in specifications. This completes construction of the visible specification. A tool such as the specification analyzer may be used to ascertain the correctness of the specifications.

4. The abstract data type QUEUE is represented as a record type structure in the private part of the QUEUE_MANAGER package. This is the first and most critical decision in the implementation. The values of this structure are subject to a type constraint which expresses a major decision about its use in this particular implementation.

5. The bodies of the concepts are implemented and tested for correctness.

6. The rest of operations are implemented.

The QUEUE_MANAGER package specification and body are now shown below. Note that certain annotations have been named. This method of naming annotations is not part of the Anna language definition, but an extension provided by the Anna Consistency Checking System. For more details regarding naming of annotations, see C.3.1.

```
generic
   type ELEMENT is private;
   MAX:POSITIVE;
package QUEUE_MANAGER is
   type QUEUE is private;
   EMPTY, FULL : exception;

   -- The following are the definitions of the concepts used in
   -- specifications. Note that IS_EMPTY and IS_FULL are
   -- defined in terms of LENGTH.

   --: function IS_MEMBER(E:ELEMENT;Q:QUEUE) return BOOLEAN;
   function LENGTH(Q:QUEUE) return INTEGER;
   function TOP(Q:QUEUE) return ELEMENT;
   function IS_EMPTY(Q:QUEUE) return BOOLEAN;
   --| <<SPEC_IS_EMPTY>>
   --| where
   --|     return LENGTH(Q) = 0;
   function IS_FULL(Q:QUEUE) return BOOLEAN;
   --| <<SPEC_IS_FULL>>
   --| where
   --|     return LENGTH(Q) = MAX;

   -- The following are the remaining operations of the pack-
   -- age. These are specified by the concepts defined earlier.
```

```
   procedure INSERT(E:ELEMENT;Q:in out QUEUE);
   --| <<SPEC_INSERT>>
   --| where
   --|     IS_FULL(Q) => raise FULL,
   --|     raise FULL => Q = in Q,
   --|     out(LENGTH(Q) = LENGTH(in Q)+1),
   --|     out(IS_MEMBER(E,Q));

   procedure REMOVE(E:out ELEMENT;Q:in out QUEUE);
   --| <<SPEC_REMOVE>>
   --| where
   --|     IS_EMPTY(Q) => raise EMPTY,
   --|     raise EMPTY => Q = in Q,
   --|     out(LENGTH(Q) = LENGTH(in Q)-1),
   --|     out(E = TOP(in Q));

private
   type QUEUE_ARRAY is array(INTEGER range <>) of ELEMENT;
   type QUEUE is record
      STORE:QUEUE_ARRAY(1..MAX);
      IN_PTR,OUT_PTR:INTEGER range 1..MAX := 1;
      SIZE:INTEGER range 0..MAX := 0;
   end record;
   --| <<QUEUE_INVARIANT>>
   --| where
   --|     in out Q:QUEUE =>
   --|         (Q.IN_PTR-Q.OUT_PTR-Q.SIZE) mod MAX = 0;
end QUEUE_MANAGER;


package body QUEUE_MANAGER;

   --: function IS_MEMBER(E:ELEMENT;Q:QUEUE) return BOOLEAN is
   --:     I:INTEGER := Q.OUT_PTR;
   --: begin
   --:     if IS_EMPTY(Q) then
   --:         return FALSE;
   --:     end if;
```

```
--:     loop
--:        if Q.STORE(I) = E then
--:           return TRUE;
--:        end if;
--:        I := I mod MAX + 1;
--:        if I = Q.IN_PTR then
--:           return FALSE;
--:        end if;
--:     end loop;
--: end IS_MEMBER;

function LENGTH(Q:QUEUE) return INTEGER is
--| <<BODY_LENGTH>>
--| where
--|     return Q.SIZE;
begin
   return Q.SIZE;
end LENGTH;

function TOP(Q:QUEUE) return ELEMENT is
--| <<BODY_TOP>>
--| where
--|     return Q.STORE(Q.OUT_PTR);
begin
   if IS_EMPTY(Q) then
      raise EMPTY;
   else
      return(Q.STORE(Q.OUT_PTR));
   end if;
end TOP;

function IS_EMPTY(Q:QUEUE) return BOOLEAN is
--| <<BODY_IS_EMPTY>>
--| where
--|     return Q.SIZE = 0;
begin
   return Q.SIZE = 0;
end IS_EMPTY;
```

```
      function IS_FULL(Q:QUEUE) return BOOLEAN is
      --| <<BODY_IS_FULL>>
      --| where
      --|     return Q.SIZE = MAX;
      begin
         return Q.SIZE = MAX;
      end IS_FULL;

      procedure INSERT(E:ELEMENT;Q:in out QUEUE) is
      begin
         if IS_FULL(Q) then
            raise FULL;
         end if;
         Q.STORE(Q.IN_PTR) := E;
         Q.SIZE := Q.SIZE+1;
      end INSERT;

      procedure REMOVE(E:out ELEMENT;Q:in out QUEUE) is
      begin
         if IS_EMPTY(Q) then
            raise EMPTY;
         end if;
         E := Q.STORE(Q.IN_PTR);
         Q.OUT_PTR := Q.OUT_PTR mod MAX + 1;
         Q.SIZE := Q.SIZE-1;
      end REMOVE;

   end QUEUE_MANAGER;
```

5.6.2 The Debugging Session

A session with the Anna Consistency Checking System aimed at testing and debugging the QUEUE_MANAGER package is now described. The session consists of six *interactions* in which tests are made, results are analyzed, and further actions are taken. Each interaction demonstrates how formal specifications are used in the two-dimensional pinpointing methodology.

INTERACTION 1

Test: A queue variable Q0 and an element variable E0 is declared. All private part annotations and package body annotations are suppressed to test the behavior of the package body for consistency with visible specifications. The following test sequence of three calls to package operations is then performed:

```
INSERT(1,Q0); INSERT(2,Q0); REMOVE(E0,Q0);
```

Result: No Anna violation occurred at the visible level. But there is an inconsistency with the intuitive intention of the program since 2 was removed from Q0, whereas the first element inserted was 1. The *Program I/O* window in Figure 5.6 illustrates this interaction.

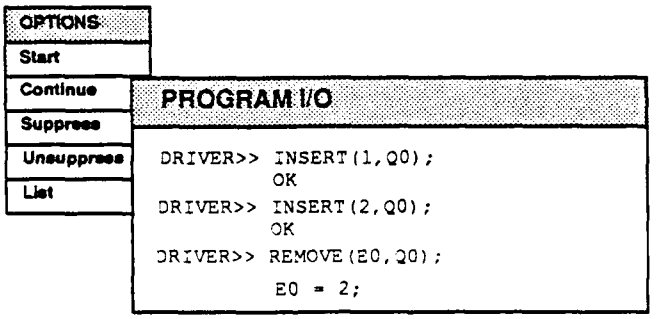


Figure 5.6: Result of Interaction 1

Explanation: This happened because the visible specifications do not express all the intuitive requirements of queues—clearly something has been forgotten. At this stage, an unexpected result has occurred at the end of the sequence:

```
INSERT(1,Q0);INSERT(2,Q0);REMOVE(E0,Q0);
```

where the interaction level is the package visible level. The region of suspicion is the visible specifications of INSERT and REMOVE, the data level, and the bodies of these package operations. See Figure 5.12 in page 130.

Guideline: *Pinpointing the region of suspicion at the visible level.*
Starting with the operation that was violated, and working back through the test sequence:

1. Express the violated intuitive requirement as a new formal annotation.

2. Consider how the previous operations of the test influence the violated requirement. If there seems to be any missing specification of the previous operations that is related to the violated one, express it formally. Missing specifications at the debugging stage are often invariants of visible operations.

Action and Justification: The first general guideline stated in page 111 is followed. One has to consider why this inconsistency occurred and look for missing visible specifications. The first guess that comes to mind is:

REMOVE did not return the value TOP(Q0)

However, the visible specification of REMOVE does contain:

```
--| out (E = TOP(in Q));
```

If this was violated, an inconsistency would have be reported by the Anna Consistency Checking System. However, no inconsistency was reported. Hence, the guess made above is wrong since TOP is assumed to be correct. The next guess that comes to mind is:

The value TOP(Q0) was changed in one of the executions of INSERT

At this stage, consider the expected behavior of INSERT with respect to the concept TOP. It looks obvious that TOP(Q) for a non-empty Q must remain invariant under an INSERT operation, and that this is missing in the visible specification of INSERT. This missing specification is formally expressed as:

```
procedure INSERT(E:ELEMENT;Q:in out QUEUE);
--| <<SPEC_INSERT>>
--| where
--|     out(TOP(Q) = if IS_EMPTY(inQ) then
--|                      E
--|                  else
--|                      TOP(in Q)
--|                  end if);
```

This is added to the specification of INSERT and the same test sequence is rerun (see Interaction 2), in order to determine whether or not the second guess is correct.

Remarks:

1. When the two-dimensional pinpointing methodology is applied at the visible level, it is important not to reason about the cause of an inconsistency based on a particular implementation detail such as elements being placed in some positional order in a common data structure as an array or a linked list. At the visible level of QUEUE_MANAGER, all that is available are the visible operations of this package, and the abstract relations between these operations. For example, it cannot be assumed that the value of TOP(Q0) is the first element in an array.

2. It is possible to prove formally that the newly added specification of INSERT must be violated by the sequence test.

3. Such invariants as the newly added specification are often overlooked.

4. If a standard debugger were used, the values of low level variables in the package body would have to be printed, and based on this, the visible level concepts would have to be manually reconstructed. The Anna Consistency Checking System automatically performs these operations, once the guesses are expressed formally.

Other Possible Actions: Another possibility would be to add:

```
procedure INSERT(E:ELEMENT;Q:in out QUEUE);
--| <<SPEC_INSERT>>
--| where
--|     out (for all E1:ELEMENT =>
--|         (IS_MEMBER(E1,Q) <-> E1=E or IS_MEMBER(E1,in Q)));
```

This might be the result of a guess that INSERT may be deleting some elements from the queue. However this specification does not constrain against INSERT changing the value of TOP, which is one of the observations made in this interaction. Hence this choice is not as good as the one that was actually made.

INTERACTION 2

Test: The same sequence is run again.

Result: An inconsistency is reported, with respect to the newly added specification, by the Anna Consistency Checking System at the end of execution of the second operation, INSERT(2,Q0). See Figure 5.7.

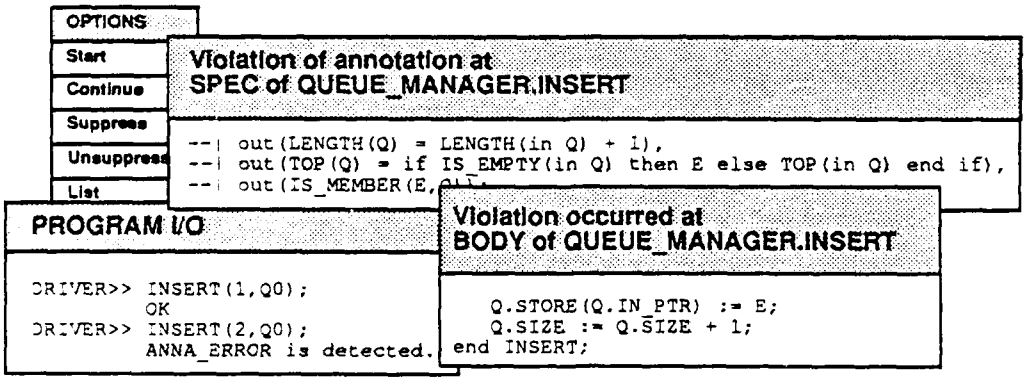


Figure 5.7: Result of Interaction 2

Explanation: As expected, INSERT has an additional effect (beyond what is specified). The window displaying the violated annotation shows that the value of TOP is not invariant

under INSERT's into non-empty queues. Note that it is the second INSERT operation that caused the inconsistency.

The length of the sequence where a violation occurred has decreased. As a result, the region of suspicion is now reduced to the visible specification of INSERT, the data level, and the body of INSERT. See Figure 5.12 in page 130. This, however, does not mean that REMOVE is correct. It simple means that an inconsistency between the specifications and the program text exists in the current region of suspicion.

Guideline: *Pinpointing the region of suspicion at the data or body level.*

1. Test the subprogram against the existing lower level annotations such as data invariants in the private part or body, subprogram body annotations, etc.

2. Add an annotation of the subprogram body which is a transformation of the visible specification that was violated. In transforming a visible annotation into an equivalent hidden one, abstract variables are replaced by their lower level (hidden) representations. Such an annotation can be used as a starting point for further pinpointing at the body level. It can be used in other situations, too. See the guidelines in Interaction 3.

3. Add more detailed subprogram annotations to the subprogram body. A good candidate is an annotation of the subprogram body referring to hidden components or variables; such detailed annotations are not expressible as subprogram annotations at the visible level.

Action and Justification: The second general guideline stated in page 111 is followed. The aim now is to reduce the region of suspicion, interacting at the next lower level in the package structure, which is the data level. For this purpose, QUEUE_INVARIANT which is in the private part of the package is un-suppressed. The same test sequence is run again (see Interaction 3). QUEUE_INVARIANT must be satisfied by all variables (and parameters) of a QUEUE record whenever a package operation terminates normally. Note that at the (lower) data level the structure and components of queues are visible.

Remarks:

1. With a standard debugger, the interaction with the program is performed at the same level as the low-level information shown in the window displaying the location of a

violation. On the other hand, in this methodology, the interaction with the program is at the same level as the high-level specification shown in the window displaying a violated annotation. At this stage of the two-dimensional pinpointing process, it is not clear yet as to how the low-level infor nation relates to the high-level specification.

2. The same remark as stated in Interaction 1 is applicable here. With a standard debugger, the values of low-level variables Q0.IN_PTR, Q0.OUT_PTR and Q0.SIZE have to be printed before and after each execution of the operation INSERT, in order to check whether QUEUE_INVARIANT is satisfied or not. However, the Anna Consistency Checking System does this automatically.

INTERACTION 3

Test: The same sequence is run again.

Result: The Anna Consistency Checking System detects a violation at the first call of INSERT as shown in Figure 5.8.

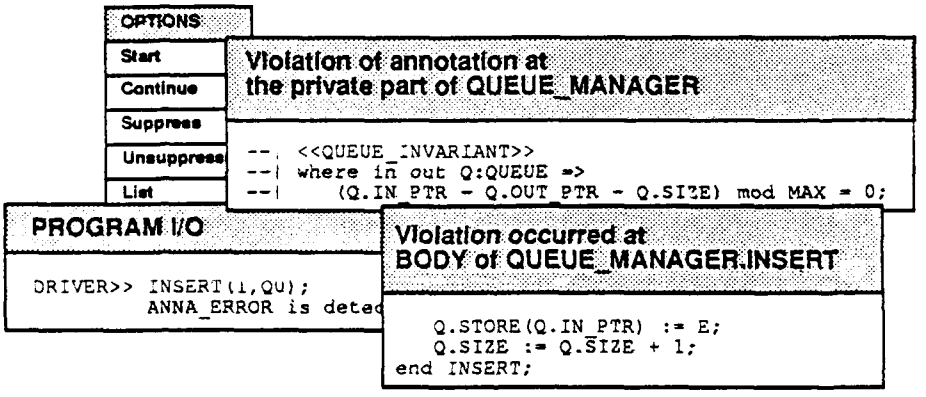


Figure 5.8: Result of Interaction 3

Explanation: This means that after an element was inserted, the invariant condition among Q.IN_PTR, Q.OUT_PTR, and Q.SIZE was violated. Notice that both dimensions, the *length* of the violated sequence and the *level* of the violated specification are reduced. The region of suspicion has been reduced to the data level and the body of INSERT. See Figure 5.12 in page 130.

Guideline: *Pinpointing the region of suspicion within a single subprogram.*

1. Add a subprogram annotation to the subprogram body which is a transformation of the visible specification that was violated. This annotation can be used for further pinpointing as well as for repair.

2. Add more detailed subprogram annotations to the subprogram body.

3. Add statement annotations such as loop invariants or assertions within the code of the subprogram body.

4. At the subprogram body level, a considerable number of detailed annotations are available. Thus, it is sometimes advisable to rewrite the body that is under suspicion, using these annotations, rather than make an attempt to further pinpoint the location of the problem.

Action and Justification: The length of the sequence is reduced to one operation—INSERT. The decision taken at this stage is to repair the body of INSERT using a goal-oriented approach. The body of INSERT must achieve the following four goals as a result of updating Q.IN_PTR, Q.OUT_PTR, and Q.SIZE:

1. **out**(Q.STORE(**in** Q.IN_PTR) = E)

2. **out**(Q.SIZE = **in** Q.SIZE + 1)

3. **out**(Q.IN_PTR = **in** Q.IN_PTR **mod** MAX + 1)

4. <<QUEUE_INVARIANT>>

Since the fourth goal was violated, the body did not achieve it. A method of fixing the body has to be found so that it achieves the fourth goal. Looking at the body, it is obvious that the body achieves the first and second goals, but not the third one. It can be informally concluded that if an assignment statement is added to satisfy the third goal, then the fourth goal is also met. Hence, the following assignment statement is added to the body of INSERT:

```
Q.IN_PTR := Q.IN_PTR mod MAX + 1;
```

Other Possible Actions:

1. The package can be run on the sequence again after adding the above goals as subprogram annotations.

2. The correctness of this repair can be checked formally by proof methods. For example see [34].

INTERACTION 4

Test: The same sequence is run again, checking specifications at all levels.

Result: The Anna Consistency Checking System detects another violation at the visible level as shown in Figure 5.9.

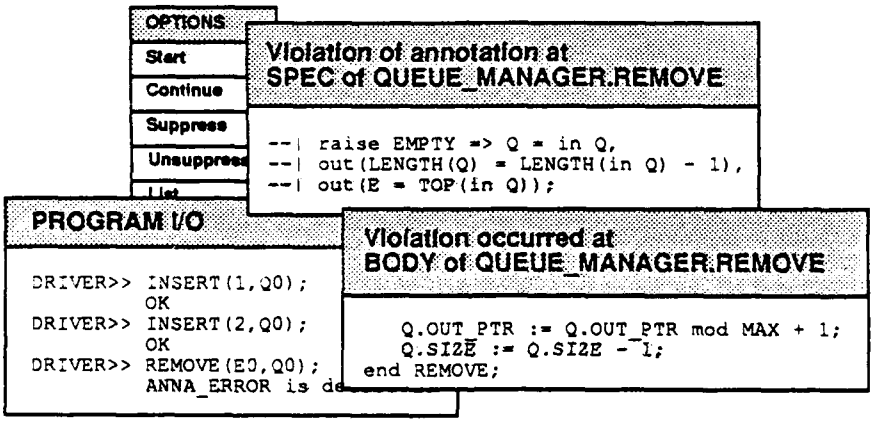


Figure 5.9: Result of Interaction 4

Explanation: This test serves two purposes:

1. Running INSERT(1, Q0); INSERT(2, Q0) against the specifications at all levels in order to find out whether or not the repair made in the previous interaction is correct on this test.

2. Running INSERT(1, Q0); INSERT(2, Q0); REMOVE(E0, Q0) against visible specifications for further testing.

The result shows that INSERT, as repaired, is consistent with all existing specifications on this particular test sequence, but that another inconsistency occurred with a visible level specification. The region of suspicion is the visible specifications of INSERT and REMOVE, the data level, and the bodies of these procedures. See Figure 5.12 in page 130.

Action and Justification: The test sequence leading to a new violation and the level of the violated annotation is the same as in Interaction 1. However, INSERT is consistent on this test with all existing annotations at all levels. A judgement is made that the specifications of INSERT are complete, i.e., there does not seem to be any further missing specifications. (there is confidence in the correctness of INSERT). Therefore, following the third general guideline stated in page 112, the region of suspicion is now reduced to the visible level of REMOVE, the data level, and the body of REMOVE. The aim now is to reduce the region of suspicion to the data level and the body of REMOVE. At this stage, a strict adherence to two-dimensional pinpointing requires first attempting to find an inconsistency at the data level. The explanation of this process is omitted for brevity. An attempt is made now to further reduce the region of suspicion to the body of REMOVE. The detailed body annotations for REMOVE must now be provided. The violated visible annotation is:

```
--| out(E = TOP(in Q));
```

This specifies exactly what the value of E in REMOVE should be. The violated visible specification is transformed into a body annotation of REMOVE, using the following body level specification of TOP:

```
--| return Q.STORE(Q.OUT_PTR);
```

The resulting specification is:

```
procedure REMOVE(E:out ELEMENT;Q:in out QUEUE)
--| <<BODY_REMOVE>>
--| where
--|     out(E = in (Q.STORE(Q.OUT_PTR)));
...
```

This transformed annotation describes exactly how E should be updated in the body of

REMOVE. This transformed specification is added to the body of REMOVE and the same sequence is run again (see Interaction 5).

INTERACTION 5

Test: The same sequence is run again.

Result: The Anna Consistency Checking System detects a violation of the new body annotation of REMOVE, as shown in Figure 5.10.

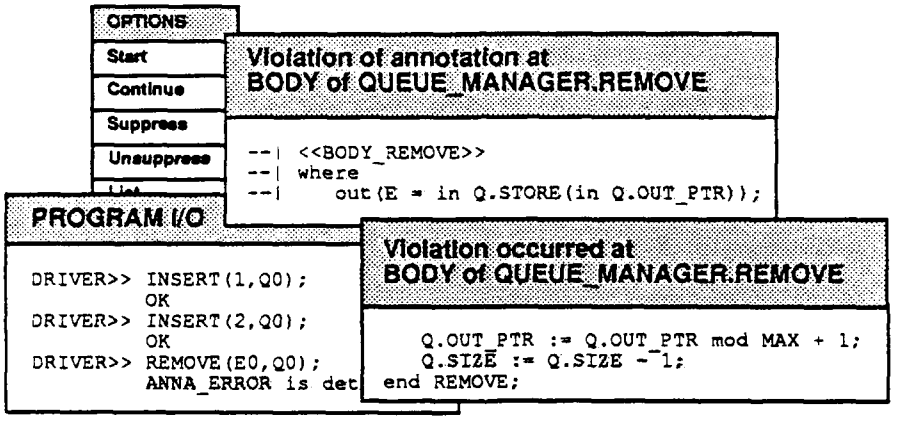


Figure 5.10: Result of Interaction 5

Explanation: The region of suspicion has been reduced to the body of REMOVE. See Figure 5.12 in page 130.

Action and Justification: The annotation BODY_REMOVE specifies the exact component in Q.STORE to be returned as the value of E. An inspection of the body of REMOVE shows that the following repair is needed: IN_PTR needs to be replaced by OUT_PTR in the assignment statement:

```
E := Q.STORE(Q.IN_PTR);
```

Other Possible Actions: The correctness of this repair could also be checked formally by proof methods.

INTERACTION 6

Test: After changing the body of REMOVE as mentioned in Interaction 5, the same sequence is run again.

Result: This time, no violation is detected. See Figure 5.11.

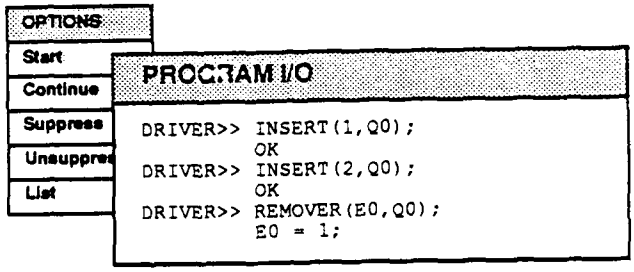


Figure 5.11: Result of Interaction 6

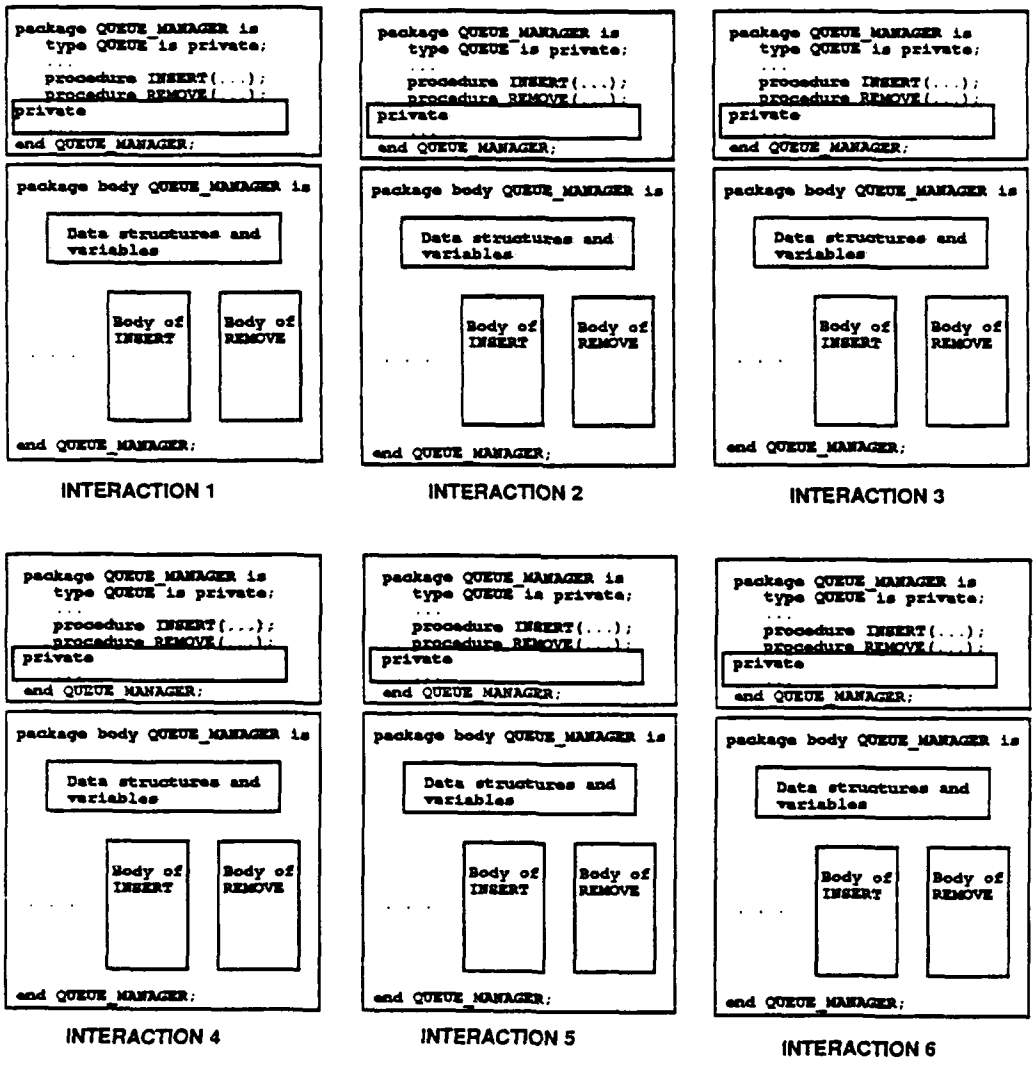


Figure 5.12: The Region of Suspicion at Each Interaction

Chapter 6

Conclusions

The Anna Consistency Checking System as described in this thesis has been implemented. Various subset restrictions are mentioned in Appendix C. This system has been used extensively both in and outside Stanford. The debugging experiments that resulted in the two-dimensional pinpointing methodology demonstrates clearly the usefulness of such a system. The Anna Consistency Checking System has also proved useful in the maintenance of large programs where consistent updates to the program are usually quite difficult. For example, an implicit assumption made at some point about the program may be violated by an inconsistent update many months later (possibly by another programmer). Writing out this assumption as an annotation and running the updated program through the Anna Consistency Checking System reveals the problem with the update immediately. Variations of this system could be useful in program optimization and program verification. The database group at Stanford has used the Anna Consistency Checking System extensively in some of their projects. Their code represents some of the largest programs that have been run through this system—some modules were about 5000 lines long.

The current system has a deficiency in that it does not understand all Ada programs. For example, array generic formal parameters and Ada aggregates are not handled. Hence, there are occasions when an Ada program with very simple annotations may not be completely be analyzed by this system because certain Ada constructs used are outside the currently handled subset. Work is currently underway to extend the capabilities of the Anna Consistency Checking System so that it can handle any Ada program. The system is also currently being updated to handle more Anna constructs. For example, *modified subtype annotations* will soon be handled. Concurrent checking of generalized assertions

is fully implemented, while a similar implementation for algebraic specification checking is still underway.

The fact that the target language of the Anna Transformer is Ada causes some problems for the Anna Transformer. It is sometimes quite difficult to create an Ada program that performs all necessary checks correctly. For example, a transformation is performed to make equality operators directly visible even when their corresponding types are either visible indirectly, or visible through the use of a subtype. This is required since there are situations in Anna where the equality operator is required for the purpose of a transformation, but would otherwise not be available due to Ada's visibility rules. Given all this, if the programmer tries to explicitly redefine the equality operator, the program becomes semantically incorrect for the Anna Transformer has already defined one.

The used interface to the Anna Consistency Checking System is constantly undergoing change. The window based user-interface described in this thesis already has the capability to invoke editors and to customize fonts, colors, menus, etc. In the long run it is hoped that the system will be completely be revamped to run incrementally. The interactions described in Chapter 5 can then take place in real-time, rather than having to perform a complete retransformation and recompilation between every interaction.

A major project that has been proposed is to extend the Anna Consistency Checking System to handle annotations on access types and objects. Anna defines the notion of a *collection* and provides a powerful set of primitives using which they can be annotated. The current inclination towards the methodology to be used for this purpose is to axiomatize collections and access types (using algebraic specifications), and to treat operations on access types and objects as operations of this axiomatized abstract data type. A straightforward application of the algebraic specification checking methodology will then work. This approach solves the problem of dynamic aliasing that access types and objects suffer from. Using the terminology of Chapter 4, all access variables accessing the same objects will be lumped together into the same equivalence class of objects.

A topic for future research is in attempting to use checking functions for other purposes than just to check the truth or falsity of annotations. One possibility is for the checking function to perform proofs, thereby generalizing the results of the checks it has already performed to a larger domain. In the best case, these "proving functions" could greatly aid in verifying the program. But at worst, the "proving functions" can be used to perform optimizations on the amount of checks performed. These kind of optimizations would be

performed dynamically. Static optimization by performing as much proof as possible before the program starts executing is also something to be looked into.

One obvious topic for future research in the case of algebraic specification checking is in extending the subset of the Chromatic Theorem Prover and in trying to theoretically characterize the limitations more precisely than has been done in this thesis. Another future research topic is in cleaning up the definition of algebraic specification checking so as to include cases like the one in Example 4.1.

The interaction between the generalized assertion checking and the algebraic specification checking systems has to be studied in detail. There are situations where an assertion check may be more comprehensive if previous results of algebraic specification checking is taken into account. This problem is not as difficult as some of the other possible extensions to this research, but still needs some time to be devoted to it to come up with a clean interaction between the two checking subsystems.

Finally, it is hoped that the experience of tool development for Anna will result in the design of better specification languages and more powerful tools in the future.

Appendix A

An Overview of Ada

The *Ada* programming language was developed on behalf of the U.S. Department of Defense to help control the cost of software for computers embedded in larger systems. The Defense Department sought a standard programming language both because it was using too many specialized languages and because the languages it was using were technically obsolete. The first step was to identify and refine the requirements for the standard language. After determining that these requirements were feasible, but not met by any existing programming language, the Defense Department awarded contracts for the design of a new language. The language Ada, which was named after Ada Augusta, the Countess of Lovelace (considered to be one of the world's first computer programmers), was the winning design in a competition that started off with seventeen proposals.

Ada is a block-structured language, based on Pascal, offering features such as *strong typing*, *subprograms*, *packages* for encapsulation of information, *tasks* for concurrency, *separate compilation* for modular programming, *exceptions* for error processing and *generic units* to write general-purpose reusable components. The remaining sections of this Appendix explains Ada in sufficient detail to be able to understand the examples in this thesis. A knowledge of Pascal, or any other similar programming language is assumed. For a more extensive coverage of Ada, please refer to [2,16].

A.1 Ada Programs

An Ada program (or Ada program library) is a collection of files (called *compilations*), each of which contains a collection of modules (called *compilation units*). Each compilation unit

is comprised of two parts—the *context clause* which lists the compilation units that this unit is dependent on, and the remaining part which is either a subprogram, a package or a generic unit. An example of a compilation unit that is a package is shown below:

```
-- A Queue Package
with TEXT_IO, LISTS;
package QUEUE is
   procedure INSERT(E:in ELEMENT);
   function REMOVE return ELEMENT;
end QUEUE;
```

The first line in this example is a comment. Everything between -- and the end of the line is treated as a comment in Ada. The second line is the context clause which says that the package QUEUE depends on the compilation units TEXT_IO and LISTS.

Ada programs can be visualized as directed graphs (called *dependency graphs*), the nodes of these graphs corresponding to the compilation units and the edges corresponding to the dependency relations specified by the context clauses. All compilation units must be compiled before the compilation units dependent on them, and hence dependency graphs have to be acyclic. Any compilation unit that is a subprogram can be chosen to be the *main program* (the program starts execution at the main program). Available to all compilation units are a set of *predefined* entities. Here is where types like BOOLEAN, INTEGER and STRING, and subprograms like the boolean and integer operations are defined. Also available is a library of predefined compilation units which offer useful facilities such as input-output. To use these predefined compilation units, they must be included in the context clauses of the compilation units that are dependent on them.

Conceptually, there is a predefined package called STANDARD in which all the predefined entities and all the compilation units in the dependency graph are considered to be declared. This conceptualization is useful in understanding the visibility rules between compilation units.

A.2 Subprograms

A subprogram is an abstract operation defined by the user. In Ada, there are two kinds of subprograms—the *procedure* and the *function*. A procedure takes a list of parameters as

input, operates on them and returns this list of parameters possibly with some updates. A function takes a list of parameters as input, and uses them to evaluate a result which is returned as the value of the function. Subprograms occur as declarations and the region of visibility starts at the point of occurrence of the subprogram and continues until the end of the construct within which they are declared. As in other block structured languages, only the name and the parameters (and the return type in the case of functions) of the subprogram are visible outside the subprogram. If a subprogram is a compilation unit, then it is visible to all the other compilation units that include the subprogram in their context clause. Procedures and functions are invoked by procedure calls and function calls respectively. A procedure call is a statement, while a function call is an expression whose value is what the corresponding function returns. Parameters can be passed in one of three possible modes: **in**—The subprogram has to treat such a parameter as a constant, and therefore can only read its value. Any expression can be passed as a parameter in this manner; **in out**—The parameter is a variable, and the subprogram is permitted to both read and update its value; and **out**—The parameter is a variable and the subprogram can only update its value. Certain attributes of **out** parameters can however be read. Functions can only have **in** parameters. The *return statement* in Ada is used to return control from subprograms. In the case of functions, this statement must include an expression that becomes the value of the function. An example of a procedure and a function is now shown below:

```
procedure EXCHANGE(X,Y:in out INTEGER) is
   TEMP:INTEGER;
begin
   TEMP := X;
   X := Y;
   Y := TEMP;
end EXCHANGE;


function SQR(X:in FLOAT) return FLOAT is
begin
   return X*X;
end SQR;
```

Examples of calls to these subprograms are shown below:

```
EXCHANGE(A,B);
S := SQR(5.5);
```

It is sometimes useful in Ada to split a subprogram into its *specification* (its interface with the rest of the program) and its *body* (its implementation). Some examples where such splitting is useful are in mutually recursive subprograms and in packages where it is necessary to separate the interface of the package from its implementation. When a subprogram is split up, its body remains the same as before, and the specification is just that portion of the subprogram where its name and parameters are described. The specifications of the subprograms in the above examples are shown below:

```
procedure EXCHANGE(X,Y:in out INTEGER);
function SQR(X:in FLOAT) return FLOAT;
```

In Ada, one can have more than one subprogram with the same name visible at a time. This is possible when these subprograms can be distinguished by the types of their parameters (and their result types in the case of functions). This phenomenon is called *overloading.* An example of overloaded functions is shown below:

```
function MOD(X:in INTEGER) return INTEGER;
function MOD(X:in FLOAT) return FLOAT;
```

The call MOD(−5) will refer to the first of the above functions, while MOD(−5.0) will refer to the second function. The process of determining the correct subprogram being referred to is termed *overload resolution.*

A.3 Packages

A package is a construct which can be used for encapsulation of information. Packages consist of two parts—the specification which defines the interface of the package with the rest of the program, and the body which implements the specification. Typically, a package specification contains declarations of types, constants, subprograms and exceptions, although any form of declaration can be included in the specification. The package body contains

the implementation of the declarations in the specification (sometimes no implementation is necessary, in which case, the package body can be omitted), and also contains (if necessary) a sequence of statements to initialize the package. An example of a package specification and body is shown below:

```
-- This is the package specification.
package STACK is
   procedure PUSH(E:in ELEMENT);
   function POP return ELEMENT;
end STACK;

-- This is the package body.
package body STACK is
   STORE:array(1..100) of ELEMENT;
   TOP:INTEGER range 0..100;

   procedure PUSH(E:in ELEMENT) is
   begin
      TOP := TOP+1;
      STORE(TOP) := E;
   end PUSH;

   function POP return ELEMENT is
   begin
      TOP := TOP-1;
      return STORE(TOP+1);
   end POP;

begin
   -- These are the initialization statements.
   TOP := 0;
end STACK;
```

Packages can occur wherever subprograms can occur and the visibility of package names follow the same rules as that of subprogram names. In addition all entities declared within the package specification are also visible at the same places the package name is visible. However, these entities have to be referred to using a special notation in which the package

name occurs as the prefix. The example below shows the use of this notation in calling the subprograms in the above package:

```
STACK.PUSH(E1);
E2 := STACK.POP;
```

This notation is sometimes cumbersome, and therefore a declaration called the *use clause* is provided which causes all entities in the specified packages to become *directly* visible. An example of this declaration is shown below:

```
use STACK;
```

It is sometimes necessary to export a type from a package, while at the same time not revealing the structure of the type. To do this, Ada provides a *private type declaration.* The type is declared to be private, and at the end of the package specification is a *private part* where the structure of the type is defined. A variation of the earlier stack package with private type declarations is shown below:

```
package STACK_PACKAGE is
   type STACK is private;
   procedure PUSH(S:in out STACK;E:in ELEMENT);
   procedure POP(S:in out STACK;E:out ELEMENT);
private
   type ELEMENT_ARRAY is array(1..100) of ELEMENT;
   type STACK is record
      STORE:ELEMENT_ARRAY;
      TOP:INTEGER range 0..100 := 0;
   end record;
end STACK_PACKAGE;
```

In this example, the fact that STACK is a record type is unknown outside the package.

A.4 Exceptions

During execution of a program, events or conditions often occur that might be considered *exceptional.* Some examples of exceptional conditions are errors like an arithmetic overflow,

and unpredictable events like reaching the end of a file of an unknown size. While it may be often possible to insert explicit tests to handle such situations, such extra statements can quickly obscure the program's basic structure. Therefore special language constructs are desirable. Ada provides three related constructs—the *exception declaration*, the *raise statement* and the *exception handler*—to handle exceptional situations.

The exception declaration can be used to declare *exceptions*. When the program gets into an exceptional situation, it can use one of these exceptions to flag the situation. This process is termed *raising* the exception. The *raise statement* can be used to achieve this. Once an exception is raised, it is *propagated* until it reaches an appropriate exception handler. Exception handlers are associated with constructs like subprograms, packages and block statements. These are located after the last statement within these constructs. The exception handlers make a correspondence between exception names and the action to be taken if that exception is propagated to this exception handler.

The propagation of an exception takes place in the following manner: If an exception handler containing an action corresponding to the raised exception exists at the end of the current subprogram, package, block statement, etc., then control is transferred to this exception handler. If not, the current subprogram, package, block statement, etc. is left *abnormally*, and control is returned to the calling/enclosing construct. The exception is re-raised at this point. This process continues until either an exception handler which contains an action corresponding to the exception is found, or when the outermost construct is reached. If such an exception handler is found, then the specified action is taken and the program proceeds *normally* from here. Otherwise the program terminates execution. An example with these constructs is now shown below:

```
-- The exception declaration
OVERFLOW, UNDERFLOW : exception;
...
function POP return ELEMENT is
begin
   if TOP = 0 then
      -- The raise statement
      raise UNDERFLOW;
   end if;
```

```
        TOP := TOP - 1;
        return STORE(TOP + 1);
    end POP;
    ...
    procedure MAIN is
        E: ELEMENT;
    begin
        E := POP;
        ...
    -- The exception handler
    exception
        when OVERFLOW =>
            PUT("Too many elements have been pushed into the stack!");
        when UNDERFLOW =>
            PUT("You are trying to pop out of an empty stack!");
        when others =>
            PUT("Unknown exception raised!");
    end MAIN;
```

A.5 Declarations

Ada provides a wide variety of declarations, many of which will now be briefly described. These declarations can be placed in any declarative region of a subprogram, package, block statement, etc. The entities defined by these declarations are visible from the point of declaration until the end of the construct in which the declaration exists. There maybe places within this region where these entities are not directly visible, for example, there may be a nested subprogram that contains a declaration with the same name. In such a situation, the entity is considered to be hidden.

A.5.1 Type and Subtype Declarations

The type and subtype declarations in Ada will be described below. Ada uses *named type equivalence* rather than *structural type equivalence*, and hence objects of two different, but structurally similar types cannot be mixed.

Enumeration Type Declarations: This defines an *enumeration type* and a domain of values that can be represented as identifiers or characters. For example, the type declaration

```
type GENDER is (MALE, FEMALE);
```

defines a type called GENDER with two values MALE, and FEMALE in its domain. These values are referred to as *enumeration literals*. For all practical purposes, enumeration literals are treated in Ada as parameterless functions whose return type is the enumeration type.

Integer and Real Type Declarations: These declarations cause new types to be defined with the properties naturally expected of integers and reals. In the declarations, an upper and lower bound can be specified, and in addition, in the case of real type declarations, an accuracy factor can be specified. This can be in the form of a certain number of significant digits (floating point type declaration) or a certain absolute value of accuracy (fixed point type declaration). Some examples of these declarations are shown below:

```
type INDEX is range 0..100;
type SOLUTION is digits 4 range -10.0..+10.0;
type BALANCE is delta 0.01 range 0.00..10000.00;
```

Derived Type Declarations: A *derived type declaration* creates a new type based on an already existing type. Typically all properties and operations are inherited. An example is shown below:

```
type DOLLAR is new INTEGER;
```

Here DOLLAR inherits all the properties of INTEGER, but the two types are different.

Array and Record Type Declarations: These are similar to array and record declarations in other languages. However, in Ada, array types can be declared without specifying the bounds on the indices. Different variables of the array can then have different bounds which is useful in some situations. Record types can have *discriminants*, whose values can

be assigned at a later time which can cause the record to change its structure. Examples of array and record type declarations are shown below:

```
type PATTERN is array(NATURAL range <>) of CHARACTER;

type PERSON_REC(SEX:GENDER) is record
   NAME:STRING(1..10);
   AGE:INTEGER;
   case SEX is
      when MALE => WIFE_NAME:STRING(1..10);
      when FEMALE => HUSBAND_NAME:STRING(1..10);
   end case;
end record;
```

Access Type Declarations: Access (pointer) types in Ada are very similar to those in other languages. An example of such a declaration is shown below:

```
type PERSON is access PERSON_REC;
```

Subtype Declarations: *Subtype declarations* do not define a new type. Instead they alias an already existing type. The new name might however include further restrictions on the existing type. An example is shown below:

```
subtype MALE_PERSON is PERSON(SEX => MALE);
```

All objects of subtype MALE_PERSON have the type PERSON, however their discriminant values are restricted to being MALE.

A.5.2 Object Declarations

An *object* (variable or constant) is an entity that contains a value of a given type. An *object declaration* declares an object. This declaration can have many forms—the objects can be given an initial value, the types can be further constrained, the objects can be declared to be constants, etc. Some examples of object declarations are shown below:

```
I:INTEGER;
A,B:constant FLOAT := 0.0;
P:PATTERN(1..10);
JOHN:PERSON(SEX => MALE);
S:array(1..10) of INTEGER;
```

A.5.3 Renaming Declarations

A *renaming declaration* declares another name for an entity. This entity can be an object, an exception, a package or a subprogram. Renaming declarations are useful in resolving name conflicts and in acting as a shorthand. Some examples of renaming declarations are shown below:

```
J:PERSON renames JOHN;
O_FL:exception renames OVERFLOW;
package ST renames STACK;
function POP return ELEMENT renames STACK.POP;
```

A.6 Statements

A.6.1 Null Statements

A *null statement* is a dummy statement that is useful in filling gaps in the program. It has no other effect than to pass to the next action. This is how a null statement looks like:

```
null;
```

A.6.2 Assignment Statements

An *assignment statement* is of the form *variable := expression;*. The current value of the variable is replaced by the value of the expression. Some examples are shown below:

```
I := 0;
JOHN.AGE := 35;
P(1..3) := P(5..7);
```

A.6.3 If Statements

An *if statement* selects for execution one or none of the enclosed sequence of statements depending on the value of one or more of its corresponding conditions. An example is shown below:

```
if MONTH = DECEMBER and DAY = 31 then
    MONTH := JANUARY;
    DAY := 1;
    YEAR := YEAR+1;
elsif DAY = LAST(MONTH) then
    MONTH := MONTH'SUCC;
    DAY := 1;
else
    DAY := DAY+1;
end if;
```

The *elsif* and *else* parts of *if* statements are optional.

A.6.4 Case Statements

A *case statement* selects for execution one of a number of alternative sequence of statements, the chosen alternative is defined by the value of an expression. An example is shown below:

```
case MONTH is
   when FEBRUARY =>
      LAST(MONTH) := 28;
   when APRIL | JUNE | SEPTEMBER | NOVEMBER =>
      LAST(MONTH) := 30;
   when others =>
      LAST(MONTH) := 31;
end case;
```

The case statement must have a sequence of statements for each possible value of the expression. The **others** alternative may be omitted if the remaining alternatives include all possible values of the expression.

A.6.5 Loop and Exit Statements

A *loop statement* includes a sequence of statements that is to be executed repeatedly until an *exit condition* is satisfied. This exit condition is either (1) part of the loop statement: a *while* loop condition—the sequence of statements are executed until the condition evaluates to FALSE, or a *for* loop range of values—the sequence of statements is executed once for each value in this range of values; or (2) in the form of an *exit statement*—the loop terminates if the exit condition specified by the exit statement evaluates to TRUE or if the exit statement specifies no exit condition.

Loops can be named. This can be used in qualifying the *for* loop identifier, if any, and also by exit statements which can then be used to terminate execution of any loop within which the exit statement is nested. Examples of loop statements are shown below:

```
while not END_OF_FILE loop
   for I in 1..N loop
      GET(A(I));
   end loop;
   SORT(A);
```

```
        for I in 1..N loop
            PUT(A(I));
        end loop;
    end loop;

    GCD_LOOP:
        loop
            exit GCD_LOOP when X = Y;
            if X > Y then
                X := X-Y;
            else
                Y := Y-X;
            end if;
        end loop GCD_LOOP;
```

A.6.6 Block Statements

A *block statement* encloses a sequence of statements optionally preceded by a declarative part and optionally followed by exception handlers. As in the case of loop statements, block statements can also be named if desired. An example of a block statement is shown below:

```
SWAP:
    declare
        TEMP:INTEGER;
    begin
        TEMP := V;
        V := U;
        U := TEMP;
    end SWAP;
```

A.6.7 Goto Statements

In addition to the previously mentioned methods of naming loop and block statements, every statement can also be named by a label. Labels occur just before the statement it names and is enclosed by a pair of angular brackets.

A *goto statement* specifies an explicit transfer of control from this statement to a target statement named by a label. An example of naming statements using labels and using goto statements is shown below:

```
<<GCD_LOOP>>
   if X = Y then
      goto END_OF_LOOP;
   end if;
   if X > Y then
      X := X - Y;
   else
      Y := Y - X;
   end if;
   goto GCD_LOOP;
<<END_OF_LOOP>>
   null;
```

A.7 Names and Expressions

Expressions in Ada are similar to expressions of most other languages. They are made up of objects, literals, etc. which are composed to form expressions by operations like +, −, etc. and by function calls. Examples in the previous sections have already shown many different kinds of simple expressions.

Declared entities like objects, enumeration literals, functions, etc. are referred to in expressions by *names* that represent them. Names are also used to refer to *attributes* of these declared entities. Some examples of names that refer to entities declared in previous examples are given below:

```
E
-- The innermost declaration of E.

MAIN.E
-- The E declared immediately within the innermost declaration
-- of MAIN. Here MAIN happens to be a procedure.
```

```
JOHN.all
-- Here (the innermost declaration of) JOHN accesses a record
-- object, and hence this name refers to the object it accesses.

JOHN.AGE
-- The component AGE of the record object accessed by JOHN.

'a'
-- The predefined character literal 'a'.

STANDARD."+"
-- The predefined addition operation. However, this operation
-- is overloaded. The correct one must be determined from the
-- operands.

P(5)
-- The fifth component of the array P.

JOHN.WIFE_NAME(1..5)
-- Denotes a string with index ranging from 1 to 5 corresponding
-- to the first ten characters of the string component WIFE_
-- NAME of the record object accessed by JOHN. This is called
-- an array slice in Ada.

INTEGER'FIRST
-- An attribute of the predefined type INTEGER. This attribute
-- is a constant that denotes the smallest INTEGER available
-- in this implementation.
```

In addition to names, there are a few other constructs that can occur in expressions—literals, aggregates, allocators, type conversions and qualified expressions. These will now be described briefly.

Numeric literals are real numbers or integers. There are many methods of writing these literals, but these will not be explained here. A couple of simple examples follow:

```
1, 3.1415, 2.7E-1
```

The null constant is a value that an object of any access type can have. If an access object has this value, then it currently does not access any object. This constant is written as:

```
null
```

String literals are constant arrays of the predefined type CHARACTER. An example is shown below:

```
"HELLO!"
```

Aggregates are constants of array or record types. They consist of a list of values of the components of the type enclosed within parenthesis. A couple of examples are given below:

```
('A','B','C')
-- An aggregate of the array type PATTERN.

(MALE,"JOHN␣␣␣␣␣␣␣",35,"MARY␣␣␣␣␣␣␣")
-- An aggregate of the record type PERSON_REC.
```

Allocators create a new object of an *accessed* type. This can be assigned to any object that is of an access type which accesses objects of the type the allocator created. An example is shown below:

```
new PERSON_REC(MALE)
```

This allocator can be assigned to any object of a type that accesses PERSON_REC, for example it can be assigned to JOHN which is an object of type PERSON which accesses PERSON_REC.

Since Ada permits only named type equivalence, objects of structurally similar types cannot replace each other. However, Ada allows explicit type conversion from one type to another structurally similar type. For example,

```
DOLLAR(I)
```

has the type DOLLAR and the value of the INTEGER object I.

Sometimes it may not be possible to determine the type of an expression by just examining it and its surrounding context. Such expressions can explicitly be qualified with the correct type name. For example,

```
PATTERN'("HELLO!")
```

specifies that the string literal is of the type PATTERN, and not of any other similar type, for example, the predefined type STRING.

The operator symbols in increasing order of precedence is now shown:

Logical Operators:	**and**, **or**, **xor**
Relational Operators:	=, /=, <, <=, >, >=
Binary Adding Operators:	+, −, & (concatenation)
Unary Adding Operators:	+, −
Multiplying Operators:	*, /, **mod**, **rem**
Highest Precedence Operators:	**, **abs**, **not**

In addition to these, Ada has *short circuit* control forms: **and then** and **or else**. These have the same precedence as the logical operators, and have the same values as **and** and **or** respectively, except that these control forms evaluate the right operand only if the value of the left operand cannot determine the value of the whole expression.

Ada also has membership tests: **in** and **not in**. These take as their left operand an expression, and a type or range as their right operand, and return a BOOLEAN value corresponding to whether or not the expression is a member of the right operand.

Appendix B

An Overview of Anna

Anna (**ANN**otated **A**da) is a language extension of Ada to include facilities for formally specifying the intended behavior of Ada programs. Anna was designed to meet a perceived need to augment Ada with precise machine-processable annotations so that well established formal methods of specification and documentation can be applied to Ada programs. The design of Anna was initiated in 1980 by Bernd Krieg-Brückner and David Luckham. They were joined by Olaf Owe and Friedrich W. von Henke during the subsequent development stages of the Anna design. The current Anna design is based on the ANSI standard version of Ada and includes annotations for all Ada constructs except tasking.

Anna is based on first-order logic and its syntax is a straightforward extension of the Ada syntax. Most new concepts in Anna are extensions of concepts already in Ada. For example, concepts such as scope, visibility and overload resolution also apply to Anna constructs. Anna constructs appear as *formal comments* within the Ada source text (within the Ada comment framework). Therefore, from the point of view of Ada, formal comments are just comments and hence *Anna programs* (Ada programs with Anna specifications) can be accepted by Ada compilers and other Ada tools.

As was the case in Appendix A, this Appendix also explains Anna in sufficient detail to be able to understand the examples in this thesis. For a more extensive coverage of Anna, please refer to [78,79].

B.1 Anna Formal Comments

Anna defines two kinds of formal comments, which are introduced by special comment indicators in order to distinguish them from informal comments. These formal comments are *virtual Ada text*, each line of which begins with the indicator `--:`, and *annotations*, each line of which begins with the indicator `--|`.

B.1.1 Virtual Ada Text

The purpose of virtual Ada text is to define *concepts* used in annotations. Often the formal specifications of a program will refer to concepts that are not explicitly implemented as part of the program. These concepts can be defined as virtual Ada text declarations. Virtual Ada text may also be used to *compute* values that are not computed by the actual program, but that are useful in defining the behavior of the program.

Virtual Ada text is Ada text with a few minor exceptions. That is, if all the virtual Ada text formal comment indicators (`--:`) are deleted from an Anna program, then the resulting program is a legal Ada program (with the few minor exceptions referred to earlier). However, the virtual Ada text must be such that it does not modify the semantics of the *underlying* Ada program. To achieve this the following restrictions are placed on virtual Ada text:

- Virtual Ada text may not syntactically contain actual Ada text. For example, it is not permitted to enclose actual Ada statements in a virtual Ada block.
- Execution of virtual Ada text statements may not change (directly or indirectly) the values of actual Ada objects.
- Virtual Ada text declarations may not hide any actual Ada declarations.
- Execution of virtual Ada text statements may not change the flow of control of the underlying program. Thus *return*, *exit* and *goto* statements within virtual Ada text can transfer control only within the largest enclosing virtual block or body.

The STACK example of Appendix A, Page 138 is shown below augmented with a virtual concept—LENGTH. This virtual concept can be used to specify the operations PUSH and POP as will be described later.

```
package STACK is
   --: function LENGTH return INTEGER;
   procedure PUSH(E:in ELEMENT);
   function POP return ELEMENT;
end STACK;
```

B.1.2 Annotations

Annotations are constraints on the underlying Ada program. They are made up of expressions which are typically boolean-valued. The location of the annotation in the Ada program together with its syntactic structure indicates the kind of constraints that the annotation imposes on the underlying program. Anna provides different kinds of annotations, each associated with a particular Ada construct. These are annotations of objects, types and subtypes, statements, and subprograms; in addition there are *axiomatic* annotations of packages, *propagation* annotations of exceptions and *context* annotations of entity visibility.

In addition to Ada expressions, expressions in annotations can also contain (amongst others) *quantified expressions*, *conditional expressions*, *state expressions* and *Anna membership tests*. Every annotation has a region of Anna text over which it applies, called its *scope*. The scope of an annotation is determined by the Ada scoping and visibility rules based on the position of the annotation in the Anna program. For example, if the annotation occurs in the position of a declaration, its scope extends from its position to the end of that declarative region. Generally, annotations constrain all *observable states* within their scope. An observable state is one that results from either the elaboration of a declaration or by the execution of a simple statement. What this means is that annotations do not constrain intermediate program states that occur during the execution of simple statements.

The following sections describe in more detail the different kinds of Anna expressions and annotations.

B.2 Anna Expressions

B.2.1 Quantified Expressions

Both universal and existential quantifiers can be used in Anna expressions. The quantified variables are referred to as *logical variables.* Logical variables can be quantified over a range of values or over all the values of a type. Any Anna constraints on the type being quantified over also constrains the range of quantification. The range of quantification is also further constrained to only those values for which the quantified expression is defined. A few examples are now shown below:

```
for all X,Y:NATURAL => X-((X/Y)*Y) = X mod Y
-- This expression is true. Note that the range of quantification
-- for Y does not include 0.

exist X:INTEGER => X*X = 100
-- This expression is also true.
```

B.2 Conditional Expressions

Conditional expressions are expressions which can take on one of a set of values depending on the values of a set of guards. An example is shown below:

```
F(X) = if X = 0 then
           1
       elsif X = 1 then
           1
       else
           F(X-1)+F(X-2)
       end if
```

The guards are evaluated in sequence from the beginning until one of the guards evaluates to TRUE. The value of the expression corresponding to this guard then becomes the value of the conditional expression. If all guards evaluate to FALSE then the value of the expression in the *else part* becomes the value of the conditional expression. All conditional expressions must contain an *else part.*

B.2.3 State Expressions

State expressions are useful to describe values of composite objects, especially as a result of a modification to one of their components. State expressions of arrays, records and package states are described in the following paragraphs. A fourth form—state expressions of collections of access types will not be dealt with here. State expressions have the following format:

value [*modification*]

Here, *value* is of the composite type and *modification* describes a modification to one of its components. A short form is available to describe values that result from sequences of such modifications:

value [$modification_1$; $modification_2$]

is equivalent to:

value [$modification_1$] [$modification_2$]

Array and Record State Expressions

The *modification* of array and record state expressions have the following format:

component => *expression*

Examples are shown below:

```
STR[3 => 'I']
-- This expression has the value of array STR except that the
-- third component is replaced by 'I'.
JOHN[AGE => 36; WIFE_NAME => "LINDA␣␣␣␣␣␣"]
-- This expression has the value of record JOHN except that the
-- components AGE and WIFE_NAME are modified.
```

Package States and Package State Expressions

A *package state* in Anna is a value that represents the state of a package. The package state is modeled as a record whose components are the variables that are declared immediately within the body of the package. There are other components that form part of the package state like the states of nested packages, but they will not be dealt with here. Within the package body, where the details of the package state are visible, package state values can

be used in about the same way as record values. Outside the package body, package state values behave similarly to private type values. Anna defines two attributes of packages that are used in conjunction with package states. If P is a package, P'STATE denotes the *current state* of the package P. For notational convenience in annotations, the current state may be denoted by the package name itself; i.e. the attribute designator STATE may be omitted. P'TYPE denotes the implicit record type that models the package state. *package state expressions* describe the effects on the package state as a result of executing package operations.

The *modification* of package state expressions takes the form of a call to a package subprogram. The value of the expression is the value of the package state that results when the subprogram call is executed in the specified package state. Some examples are shown below:

```
STACK'STATE[PUSH(E1)]
-- The state of the package STACK as a result of executing
-- PUSH(E1) on the current state of this package.
STACK[PUSH(E2);POP]
-- Here, 'STATE has been omitted for notational convenience.
-- This expressions denotes the state of the package STACK as a
-- result of executing PUSH(E2) followed by POP on the current
-- state of this package
```

There is another useful operation on package states. The expression S.F(...) where S is a package state expression and F is a package function denotes the value F(...) returns if it is called when the package is in state S. An example is shown below:

```
STACK[PUSH(E2)].POP
```

B.2.4 Initial Expressions

An *initial expression* contains the keyword **in** (referred to as the modifier) followed by an expression referred to as the *modified expression.* Initial expressions are constants. The value of an initial expression is the value that the corresponding modified expression had when it was elaborated. Basically, during elaboration, the initial expression is replaced by the value of the expression it modifies. Some examples are shown below:

```
in X
in (X**2+Y**2)
```

B.2.5 Anna Operators

There are four new operators in Anna. They are the implication operator, the equivalence operator, Anna relational operators and Anna membership tests. The implication operator (->) and the equivalence operator (<->) have their usual meanings:

```
A -> B   -- if A then B
A <-> B  -- A if and only if B
```

The Anna relational operators feature is a syntactic short-hand to express certain commonly occurring conjunctions of Ada relations. This is best described by an example:

```
A < B <= C
```

is the same as:

```
A < B and B <= C
```

The Anna membership test (**isin**) is an extension of the Ada membership test. The Ada membership test A **in** T checks that the value A satisfies the Ada constraints on T. However, the Anna membership test A **isin** T checks that the value A satisfies both the Ada and the Anna constraints on T. Anna constraints can be placed on T using subtype annotations (described later).

B.3 Annotations

B.3.1 Object Annotations

An object annotation is a BOOLEAN expression that constrains the values of the variables occurring in this expression throughout the scope of the annotation. Object annotations can occur in declarative regions only. In the special case in which there are no variables in

the expression, the annotation is a constant and hence is really a constraint at the point of elaboration of the annotation. A typical example of this special case is an object annotation whose expression is an initial expression. There is another kind of object annotation, the *out annotation* which constrains only the point of exit from the scope. Some examples of object annotations are now shown:

```
--| CIRCUMFERENCE = 3.14159 * DIAMETER;
-- This object annotation constrains the variables CIRCUM-
-- FERENCE and DIAMETER throughout the scope of the an-
-- notation.

--| in (X > 0);
-- An object annotation with no variables (though X is a vari-
-- able, it is within an initial expression). This annotation con-
-- strains the point at which the annotation is defined.

--| out (Y = in X);
-- An out annotation which constrains the variable Y to be equal
-- to the value X had when the annotation was elaborated.
```

B.3.2 Subtype Annotations

A subtype annotation is a constraint on types and subtypes. Unlike in the case of object annotations, there can be only one subtype annotation for each type or subtype definition. An example of a subtype annotation is shown below:

```
type EVEN is new INTEGER;
--| where X:EVEN => X mod 2 = 0;
```

This annotation constrains all objects X of the type EVEN to satisfy the constraint X **mod** 2 = 0. If the subtype annotation contains any variables other than the logical variable (X in the above example), then these variables are implicitly modified by **in** (they are replaced by their values at elaboration time). Hence subtype annotations only constrain their corresponding subtypes.

B.3.3 Statement Annotations

There are two different kinds of statement annotations—*simple statement annotations* and *compound statement annotations.*

Simple Statement Annotations

Simple statement annotations are constraints on a single statement. This constrained statement is the one immediately before the simple statement annotation. The constraint imposed by the annotation has to hold when control leaves the constrained statement—i.e. it behaves like an *out annotation* on the constrained statement. If the annotation occurs at the beginning of a sequence of statements, then it constrains an implicit *null statement* just before the annotation.

Therefore, in most cases, the simple statement annotation is really a constraint that has to hold whenever control passes the point where the annotation is located. However, when the preceding statement transfers control to some other location (as is the case with *goto*, *return* and *exit* statements) then the constraint has to hold just before control is transferred by this statement. An example is given below:

```
I := 2;
while I <= N loop
   if A(I-1) > A(I) then
      EXCHANGE(A(I-1),A(I));
   end if;
   --| A(I) >= A(I-1);
   -- This constraint has to hold after the execution of the pre-
   -- ceding if statement.
   I := I+1;
end loop;
```

Compound Statement Annotations

Compound statement annotations are constraints on compound statements. The constrained statement occurs immediately after the compound statement annotation. The annotation is bound to the statement by the keyword **with** and constrains all observable

states in the compound statement—i.e. it behaves like an object annotation on the constrained statement. The previous example is shown below once again with a compound statement annotation included:

```
I := 2;
--| with 1 < I <= N+1;
-- This constraint must hold at all times within the following
-- while loop.
while I <= N loop
    if A(I-1) > A(I) then
        EXCHANGE(A(I-1),A(I));
    end if;
    --| A(I) >= A(I-1);
    I := I+1;
end loop;
```

B.3.4 Subprogram Annotations and Result Annotations

Subprogram annotations are used to describe the behavior of subprograms. They are bound to the Ada subprogram specification by the keyword **where**. They are useful in describing the input-output specifications of the subprogram. *Result annotations* are constraints on the return values of functions. Result annotations must occur immediately within a function, but its location is otherwise not restricted. It can be in the place of an object annotation, a statement annotation, or a subprogram annotation. Result annotations are distinguished by the keyword **return** and they constrain all *return statements* within their scopes. A few examples of subprogram annotations and result annotations are shown below:

```
procedure EXCHANGE(X,Y:in out INTEGER);
--| where out(X = in Y), out(Y = in X);
-- On output the value of X is the input value of Y and vice-
-- versa.
```

```
procedure PUSH(E:in ELEMENT);
--| where out(LENGTH = in LENGTH+1);
-- This procedure is from the STACK package shown earlier.
-- In addition to illustrating subprogram annotations, this also
-- shows how virtual functions can be used for annotation pur-
-- poses. The above annotation says that the execution of PUSH
-- causes the the value of LENGTH to increase by 1.

function SQRT(X:FLOAT) return FLOAT;
--| where return Y:FLOAT => Y*Y = X;
-- The value Y returned by SQRT is such that its square is equal
-- to the input parameter X.
```

B.3.5 Axiomatic Annotations

Axiomatic annotations (or package axioms) are constraints on package operations. They must occur in the package visible part. They are characterized by the keyword **axiom** followed by a sequence of BOOLEAN expressions which are usually quantified with respect to types defined in the package. Axiomatic annotations are *promises* which can be assumed wherever the package is visible; and they are also constraints on the implementation of the package. Algebraic specifications of abstract data types can be written as axiomatic annotations. The STACK package shown earlier is axiomatically specified in the following example:

```
package STACK is
   --: function LENGTH return INTEGER;
   procedure PUSH(E:in ELEMENT);
   --| where out(LENGTH = in LENGTH+1);
   function POP return ELEMENT;
   --| where out(LENGTH = in LENGTH-1);
```

```
--| axiom
--|     for all S:STACK;E:ELEMENT =>
--|         S[PUSH(E);POP] = S,
--|         S[PUSH(E)].POP = E;
-- The above two constraints have to be satisfied by all imple-
-- mentations of this package, and therefore can be assumed
-- wherever the package is visible.
end STACK;
```

B.3.6 Context Annotations

A *context annotation* constrains the use of global variables within a program unit. Context annotations take the form of the keyword **limited** followed by a list of zero or more variables. It constrains the occurrences of variables declared outside of its scope (the program unit)—only those outside variables that are listed in the annotation may occur (be used) within its scope. A context annotation on the stack package mentioned earlier is shown below:

```
--| limited to INTEGER,ELEMENT;
package STACK is
    ...
end STACK;
```

B.3.7 Exception Annotations

Exception annotations (or propagation annotations) specify the exceptional behavior of program units. There are two different kinds of exception annotations—*strong propagation annotations*, which specify the conditions under which certain exceptions should be propagated (outside the scope of the annotation); and *weak propagation annotations*, which specify what happens when an exception is actually propagated.

Strong Propagation Annotations

Strong propagation annotations specify under which conditions exceptions should be propagated. The conditions are with respect to the initial state of the scope of the annotation. If

the conditions are satisfied, then the scope of the annotation must be exited by propagating the specified exception. Again, examples follow with respect to the stack example:

```
package STACK is
   --: function LENGTH return INTEGER;
   UNDERFLOW: exception;
   ...
   function POP return ELEMENT;
   --| where LENGTH = 0 => raise UNDERFLOW;
   -- If LENGTH is 0 when POP starts execution, then it
   -- must terminate by propagating the exception UNDER-
   -- FLOW.
   ...
end STACK;
```

Weak Propagation Annotations

Weak propagation annotations specify what happens when an exception is propagated. It specifies conditions that must be satisfied if the scope of the annotation is exited by propagating one of the specified exceptions. An example is shown below:

```
procedure EXCHANGE(X,Y: in out INTEGER);
--| where raise CONSTRAINT_ERROR => X = in X and Y = in Y;
-- If for any reason, the exception CONSTRAINT_ERROR is
-- propagated out of the procedure EXCHANGE, then the pa-
-- rameters X and Y remain unchanged.
```

Appendix C

Installation Manual and User Guide

C.1 Introduction

The installation manual describes how the Anna Consistency Checking System can be installed on a machine. The machine dependent aspects of the installation process are discussed in detail.

The user guide describes how to use this system in transforming Anna specifications into checking code and then executing the transformed program with the Anna debugger. The various transformation and debug options are described. A detailed subset restriction based on the Ada and the Anna language reference manuals is also provided.

The current release of the system has been or is currently being ported to the Rational R1000[1], DEC VMS DEC-Ada[2], ROLM/DG ADE[3], Sun/3-UNIX[4] VADS[5], Sequent/DYNIX[6] (both Sequent Balance and Symmetry) VADS, and Apollo/UNIX[7] VADS and ALSYS[8] environments. The entire system has been written in Ada.

[1] Rational and R1000 are registered trademarks of Rational, Inc.
[2] DEC, VMS, and DEC-Ada are registered trademarks of Digital Equipment Corporation.
[3] ADE is a registered trademark of ROLM/DG.
[4] UNIX is a registered trademark of AT&T Bell Laboratories.
[5] Verdix and VADS are registered trademarks of Verdix Corporation.
[6] Sequent and DYNIX are registered trademarks of Sequent Computer Systems, Inc.
[7] Apollo is a registered trademark of Apollo, Inc.
[8] ALSYS is a registered trademark of ALSYS, Inc.

A discussion of some of the other related tools is excluded from this appendix. This information is however available in another more general manual [86]. The tools not described here include the *Anna Semantics Analyzer*, the *Anna Package Specification Analyzer* and *Anna Teach*—an interactive tool that teaches Anna.

C.2 Installing the Anna System

The Anna Consistency Checking System has two components—the *transformation system* and *the runtime system*. The transformation system converts the Anna specifications into Ada checking code and instruments this checking code into the underlying Ada program. The runtime system is invoked by this resulting Ada program.

C.2.1 Setting Up the Machine Dependent Parameters

All parameters that are specific to the hardware/operating system are encapsulated into one package body. The rest of the Anna Consistency Checking System access these parameters through the visible part of this package which is machine independent. The visible part of this package is `impl_depd_v.a`. There are two separate bodies provided for this package. The first body, `impl_GENERIC_b.a`, can be used as is or as a template into which machine dependent parameters are incorporated. When used as is, the Anna Consistency Checking System will assume that the machine has no directory structure, that is, all necessary files are available in one area and can be accessed by just their file names. This generic implementation of the package might not work in certain situations. For example, in *DOS*, file names can have at most eight characters, but this generic package assumes that file names can be of any arbitrary length. The second body, `impl_VADS_UNIX_b.a`, has been designed specifically for use on *UNIX* machines running the *VADS (Verdix Ada Development System)* Ada compiler.

The first thing to be done on copying the distribution tape is to implement a body for this package appropriate for the target system. The operations defined in this package are described below. All examples are with respect to the VADS/UNIX implementation.

ADA_COMPILER_AND_MACHINE_NAME:

> This is a function which returns a string that describes the environment in which the Anna Consistency Checking System is being used. The VADS/UNIX implementation returns "`VADS/UNIX`".

SOURCE_DIRECTORY:

This function returns a string which corresponds to the directory into which the distribution tape has been copied into. The Anna Consistency Checking System at Stanford resides in the directory `/anna/xform/source` and hence this function in the VADS/UNIX implementation has been setup to return a string corresponding to this directory.

XFORM_FILE_NAME_PREFIX:

To make the names of all the files generated by the Anna Transformer unique with respect to all the other files in the same directory, the Anna Transformer prefixes all its files with the string returned by this function. The VADS/UNIX implementation returns the string "`.x`". Note that in addition to being unique, these files will also be hidden on a UNIX system.

MAKE_FILE_NAME:

Given a directory name and a file name, this function returns a string that can be used by the predefined I/O operations (e.g., the operations defined in TEXT_IO) to access this file. The VADS/UNIX implementation concatenates the directory name and the file name after inserting a "/" in the middle.

GET_DIRECTORY:

Given a file name that can be used to access a file by the predefined I/O operations, this function returns a string corresponding to the directory in which this file exists. The VADS/UNIX implementation determines if the file name starts with "/". If so, then the portion of the file name until the last occurrence of "/" is returned. Otherwise, this same portion is returned after concatenating it on the left with a string that describes the current directory.

LIBRARY_DIRECTORY:

Typically, compiler vendors place the environment of their Ada systems into a set of separate directories. The Anna Consistency Checking System handles these directories differently, for example, it does not create any of its own files in such directories. This helps keep these directories clean. This function is provided to determine whether or not the specified directory contains such an environment. If so, this

function returns TRUE, otherwise it returns FALSE. The VADS/UNIX implementation at Stanford returns TRUE if the directory is either "`/usr/vads5/standard`" or "`/usr/vads5/verdixlib`".

ADA_PATH:

This function determines the current search path of the Ada system being used. This will be a list of directories excluding the current directory. This list is concatenated after appending each of the directory names with `ASCII.NULL`, and returned as a single string. The VADS/UNIX implementation returns a string corresponding to the directories displayed on typing `a.path`.

QUIT:

This is used to implement the `QUIT` command of the Anna Debugger. It is otherwise not easy to exit from a user defined Ada program at some arbitrary point during its execution. The VADS/UNIX implementation implements this procedure by calling the UNIX function `_exit`.

The main program which invokes the Anna Transformer is called `transformer.a`. This program also contains a few implementation dependent features. For example the main program in the distribution tape reads the command line and extracts the necessary parameters from there. This main program has been provided for convenience only. The distribution tapes that includes all the tools developed at Stanford includes a more comprehensive main program with a better structure. It is expected that this main program will be modified appropriately for the target machine.

C.2.2 Compiling the Anna Consistency Checking System

Once the implementation dependent parameters have been setup, replace the occurrence of `impl_VADS_UNIX_b.a` in the file `RECOMPILE` with the name of the file that contains the newly created package body. The file `RECOMPILE` will then have to be modified for the target machine. The file names, path names and the commands to the Ada environment have to be changed appropriately. This version of the Anna Consistency Checking System uses *X windows* for its user interface. Some of the path specifications in `RECOMPILE` refer to the directories in which the X windows interface resides. If X windows is not available on the target machine, then another version of the Anna Consistency Checking System is

required, which has a simple terminal oriented user-interface. These two versions will be merged soon.

After **RECOMPILE** *has been modified, the sequence of commands in this file need to be executed, possibly by submitting it as a batch job.*

This results in the creation of an executable called **xform**. Also, all the units that form the runtime system of the Anna Consistency Checking System will have been compiled.

C.2.3 Setting Up the Predefined Environment

The last step in the installation process is to setup the predefined environment. This process involves setting up the symbol table files of units like TEXT_IO, which are assumed to be already transformed when the programmer begins using the Anna Consistency Checking System. This is achieved by executing the commands in the file **RETRANSFORM**. It may be necessary to modify this file appropriately for the target machine.

Once this is done, the Anna Consistency Checking System is fully installed and ready for use. The remainder of this appendix forms the user guide for this system.

C.3 Non-Standard Anna Features

C.3.1 Annotation Names

The Anna Consistency Checking System extends the Anna language to make it possible to name annotations. These names can then be used to refer to these annotations, for example, through the Anna Debugger.

Annotation names can precede any list of annotations, and its syntax is similar to Ada label definitions. Examples of naming annotations are shown below:

```
type EVEN is new INTEGER;
--| <<EVEN_CONSTRAINT>>
--| where X:EVEN => X mod 2 = 0;
```

```
procedure INSERT(E:ELEMENT;Q:in out QUEUE);
--| <<SPEC_INSERT>>
--| where
--|     IS_FULL(Q) => raise FULL,
--|     raise FULL => Q = in Q,
--|     out(LENGTH(Q) = LENGTH(in Q)+1),
--|     out(IS_MEMBER(E,Q));
```

C.3.2 Anna Pragmas

The following list of pragmas have been setup for use with the Anna Consistency Checking System. They are not part of the Anna language definition. The first pragma in this list makes sense only when transformed using the -d option, while all the other pragmas have to be transformed using the -p option.

SUPPRESS_ANNO:

The argument to this pragma is an annotation name. It can be placed at any point where the annotation name is visible (the visibility rules are the same as for any other kind of declaration). This pragma causes the annotation to be suppressed initially.

ANNA_PARALLEL_LOG_FILE:

This pragma takes a file name as an argument. Since diagnostic information from a task performing checks concurrently with the execution of the underlying program can clutter up the screen, this option is provided to send all such information into the specified file. If this pragma is not provided, this information is sent to the standard output. This pragma has to be placed immediately after the context clause of the main program.

ANNA_REPORT_MODE:

This pragma takes one argument which can be one of REPORTING, ABORTING or IGNORING. This pragma specifies what happens when an annotation is violated. It can be located anywhere in the program and has effect on all the annotations defined from this point until the next such pragma. Regardless of the mode of reporting

errors, a diagnostic message with details of the violation is given. Depending on the reporting mode, one of the following set of actions is then taken:

- REPORTING: If the annotation is being checked sequentially, the exception ANNA_ERROR is raised immediately. If the annotation is being checked concurrently, the exception ANNA_ERROR is raised at the next checkpoint (see below for information on checkpoints). This is the default action that is taken in the absence of pragmas.
- ABORTING: The entire program is immediately aborted.
- IGNORING: The violation has no effect on the underlying program.

ANNA_PARALLEL_SCOPE:

This pragma takes an optional parameter which can be one of REPORTING, ABORTING or IGNORING. This pragma causes all annotations in its scope (according to standard scope rules) to be transformed for concurrent checking. If the parameter is provided, then the current reporting mode is changed as specified. This pragma overrides the pragma ANNA_SEQUENTIAL_SCOPE (if any) present in outer scopes. This is the default in the absence of pragmas.

ANNA_SEQUENTIAL_SCOPE:

This pragma is similar to ANNA_PARALLEL_SCOPE, except that it causes all annotations in its scope to be transformed for sequential checking. This pragma overrides the pragma ANNA_PARALLEL_SCOPE (if any) present in outer scopes.

ANNA_PARALLEL_ANNOTATION:

This pragma has to occur immediately after an annotation. It is identical to the pragma ANNA_PARALLEL_SCOPE, except that its scope includes only the immediately preceding annotation.

ANNA_SEQUENTIAL_ANNOTATION:

This pragma also has to occur immediately after an annotation. It is similar to the pragma ANNA_PARALLEL_ANNOTATION, except that it causes the immediately previous annotation to be transformed for sequential checking.

ANNA_CHECKPOINT:

This pragma takes an optional list of annotation names as its argument. All these annotation names have to be visible at the point of the pragma, and must have been transformed for concurrent checking. The pragma has to be located as part of a sequence of statements. The effect of the pragma is to suspend the underlying program when execution reaches the checkpoint until all checks on the specified annotations are completed. If no list of annotations is specified, then the underlying program is suspended at the checkpoint until all checks on every annotation being checked concurrently has been completed.

C.4 Transforming Anna Programs

In this section, the method of transforming Anna programs to Ada programs is explained. This Appendix does not, however, address issues involved in writing good Anna programs. This is discussed in [70]. To transform an Anna program, the programmer issues the command:

```
xform {options} annafile
```

`annafile.anna` is the name of the Anna source file. After transformation, the Ada file is `annafile.a` and the parser listing is given in `annafile.list`. The various options are listed below:

-o,-O:

This option is used for tracing the overload resolution processor. The trace output is dumped into a log file called `annafile.oload`. `-O` provides more comprehensive tracing than `-o`. This option is usually not very useful, unless this system is being run without the Anna Semantics Analyzer. The Anna Transformer assumes that the source file is semantically valid, that is, it has successfully gone through the Anna Semantics Analyzer. The Anna Transformer does attempt to provide some amount of diagnostic information in case of errors, the overload resolution tracing is the major source of this diagnostic information.

-m:

This option has to be used when the source file includes the main program. The Anna Transformer assumes that the last compilation unit in the source file is the main program.

-d:

This option causes the Anna Transformer to include hooks to the Anna Debugger. If this option is not used, the Anna Debugger is not invoked at runtime on the detection of an inconsistency.

-p:

This option causes the transformations to generate checking tasks instead of checking functions. Hence the checking can take place concurrently. Note that in this mode, some annotations can be checked sequentially, while others concurrently. This is achieved using the pragmas described in C.3.2.

-z:

This option is used only to transform predefined units. See the file `RETRANSFORM`.

-T:

This option invokes a special testing mode, which is useful for regression testing of the Anna Consistency Checking System. This option is usually not necessary in normal situations.

-S:

This option causes the transformations to be performed silently. There is no output to the user console from the Anna Transformer. Instead this output is redirected to a file called "`XFORM.LOG`".

Typically, an Anna program comprises of many compilation units. The order of transformation of these compilation units follow the same rules as in the case of the order of compilations of these units. Within a program, one cannot mix the options `-d` and `-p`.

C.4.1 Creating the Self-Checking Executable

Once the compilation units of the Anna program have been transformed, the corresponding Ada units are available. These Ada units are compiled as usual and then linked and loaded to get the self-checking executable.

Note that the Ada search path of Anna directories must include the directory in which the distribution tape has been copied. The Anna Transformer displays this path name, the first time it is invoked in any directory.

C.4.2 The Anna Debugger

The Anna Debugger is invoked in any Anna program in which at least one of the units has been transformed using the `-d` option. It is invoked once when the program begins execution, and then subsequently every time an annotation is violated. The unit containing the violated annotation has to be transformed using the `-d` option. This section explains the terminal oriented user interface to the Anna Debugger. The window oriented user interface provides the same functionality, and the menus, etc. are such that its use is quite obvious. Once the terminal oriented user interface is understood, the window oriented user interface can be used easily. Since the user interface module of the debugger is a separate entity, it is easy to modify it to suit the requirements of each installation. In fact, the user interface has been continuously undergoing change at Stanford.

The Anna Debugger provides various debugging facilities suited for use with a specification language like Anna. The current version does not provide any of the features provided by a conventional debugger, and therefore for the Anna Debugger to be useful, it should be used along with a conventional debugger. There are obvious disadvantages to this kind of usage. For example, one has to switch between the two debuggers, and the conventional debugger works on the transformed Ada program which is usually not very readable. To enhance the Anna Debugger to provide the functionality of a conventional debugger, it has to be made compiler-dependent and some aspects of the compiler that is chosen have to be known. Such information is usually not divulged by compiler vendors.

The Anna Debugger has an on-line help facility which can be invoked by typing "?" or "HELP" at any time. The following features are available, however, they can be used only with named annotations.

Annotations can be suppressed or unsuppressed. Suppressing an annotation means that this annotation will have no effect at all. An annotation in the unsuppressed state can have various different kinds of effects as described below. All annotations are by default unsuppressed.

Command syntax:

SUPPRESS *annotation_name*

−SUPPRESS *annotation_name*

When an annotation is unsuppressed, it can either be made to invoke the Anna Debugger

every time the annotation is violated, or not invoke the Anna Debugger. All annotations by default invoke the Anna Debugger.

Command syntax:
INVOKE *annotation_name*
−INVOKE *annotation_name*

When an annotation is unsuppressed, it can either raise the exception ANNA_ERROR when an annotation is violated, or not raise this exception. All annotations by default raise this exception.

Command syntax:
RAISE *annotation_name*
−RAISE *annotation_name*

The status of an annotation can be queried by typing:

STATUS *annotation_name*

The command LIST lists all the named annotations.
To continue execution of the Anna program, type:

CONTINUE

Finally, to exit the Anna Debugger, type:

QUIT

C.5 Subset Restrictions: The Ada Reference Manual

This section and the next explain the subset restrictions of the Anna Consistency Checking System. This section is organized based on the Ada reference manual, while the next section is organized based on the Anna reference manual. Hence, for each section in these reference manual, there is a corresponding section in either this section or the next that explains how the Anna Consistency Checking System handles this aspect of the language.

C.5.1 Introduction

Irrelevant

C.5.2 Lexical Elements

Fully implemented

C.5.3 Declarations and Types

Declarations

Irrelevant

Objects and Named Numbers

Fully implemented

Types and Subtypes

Fully implemented, except what is specifically excluded in the later sections of this chapter.

Derived Types

Fully implemented

Scalar Types

Fully implemented

Array Types

Fully implemented

Record Types

Records without discriminants are *fully implemented.* Records with discriminants are treated as if the discriminants were declared as record components themselves. Discriminant constraints are, however, fully implemented. However, record types with variant parts will not be accepted. The attribute CONSTRAINED is not implemented.

Access Types

The attributes ADDRESS and STORAGE_SIZE are not implemented.

Declarative Parts

Irrelevant

C.5.4 Names and Expressions

Names

Fully implemented, except for the following attributes: ADDRESS, CALLABLE, CONSTRAINED, COUNT, FIRST_BIT, LAST_BIT, POSITION, STORAGE_SIZE and TERMINATED.

Literals

Fully implemented

Aggregates

Not implemented

Expressions

Fully implemented

Operators and Expression Evaluation

Fully implemented

Type Conversions

Fully implemented

Qualified Expressions

Fully implemented

Allocators

Fully implemented

Static Expressions and Static Subtypes

Static expressions are not evaluated even if they are just literals. Therefore in expressions containing attributes with static arguments (e.g. FIRST(2), LAST(3–2)) all possible legal values are tried out (this is a heuristic that may not work always). An example is given below:

```
A:array (1..10,'a'..'z') of INTEGER;
I:INTEGER;

I := A'FIRST(3-2);
-- Right hand side is correctly resolved to INTEGER.

I := A'LAST(2);
-- Right hand side is incorrectly resolved to INTEGER.
```

However, the second of the two above expressions is illegal Ada and so is not expected as input. Even then, problems may arise as shown in the example below:

```
A:array (1..10,'a'..'z') of INTEGER;
function F return INTEGER;
function F return CHARACTER;

if F = A'LAST(2) then
-- The above expression, though legal, cannot be resolved.
   ...
end if;
```

Universal Expressions

Just as in the case of static expressions, universal expressions are also not evaluated, and therefore can create ambiguities similar to that shown in the previous example.

C.5.5 Statements

Fully implemented

C.5.6 Subprograms

Subprogram Declarations

Fully implemented

Formal Parameter Modes

Fully implemented

Subprogram Bodies

Fully implemented

Subprogram Calls

Named parameter association is not supported. Also, *in out* and *out* parameters having the form of a type conversion are not supported.

Function Subprograms

Fully implemented

Parameter and Result Type Profile - Overloading of Subprograms

Fully implemented

Overloading of Operators

Fully implemented

C.5.7 Packages

Package Structure

Fully implemented

Package Specifications and Declarations

Fully implemented

Package Bodies

If the *transformation* of a package specification causes the generation of subprogram declarations in the package specification, then a body must be provided to generate the subprogram bodies in. If the original package specification does not require a body, an empty body can be provided.

Private Types and Deferred Constant Declarations

Fully implemented

Example of a Table Management Package

Irrelevant

Example of a Text Handling Package

Irrelevant

C.5.8 Visibility Rules

Declarative Region

Fully implemented

Scope of Declarations

Fully implemented

Visibility

Fully implemented

Use Clauses

Fully implemented

Renaming Declarations

Not implemented

The Package Standard

Fully implemented

The Context of Overload Resolution

The fact that a type is private or limited private is not considered during overload resolution.

C.5.9 Tasks

Nothing except the delay statement is implemented.

C.5.10 Program Structure and Compilation Issues

Compilation Units - Library Units

Fully implemented

Subunits of Compilation Units

Not implemented

Order of Compilation

The order in which compilation units are transformed follow the same rules as for the order of compilation. All units must be transformed first and then compiled.

The Program Library

The program library follows the same structure as the library maintained by the Verdix Ada Compiler. No special library set-up commands are required, all necessary information is inherited from the Verdix library. If the working directory does not have a Verdix library set up, then the standard default paths are assumed.

Elaboration of Library Units

Fully implemented

Program Optimization

Irrelevant

C.5.11 Exceptions

Fully implemented

C.5.12 Generic Units

Fully implemented except that generic formal parameters that are arrays are not allowed.

C.5.13 Representation Clauses/Implementation-Dependent Features

Not implemented

C.5.14 Input-Output

Fully implemented

C.6 Subset Restrictions: The Anna Reference Manual

C.6.1 Basic Anna Concepts

Fully implemented

C.6.2 Lexical Elements

The *Anna special characters* are not implemented.

C.6.3 Annotations of Declarations and Types

Declarative Annotations

Irrelevant

Annotations of Objects

Object annotations are not allowed within the private part of packages and within generic formal parts.

Annotations of Type and Subtype Declarations

Fully implemented

Annotations of Derived Types

Fully implemented

Operations of Scalar Types

The attribute DEFINED is assumed to be true always.

Annotations of Array Types

Fully implemented

Annotations of Record Types

Record states are not implemented.

Annotations of Access Types

Collections are not implemented.

Declarative Parts

Irrelevant

C.6.4 Names and Expressions in Annotations

Names in Annotations

None of the attributes are implemented. However, an assumption is made that all objects are defined always, i.e. the attribute DEFINED is assumed to evaluate to TRUE always. Otherwise all other forms of names are implemented unless explicitly mentioned elsewhere.

Expressions in Annotations

Fully implemented, unless explicitly mentioned elsewhere.

Operators and Expression Evaluation

Fully implemented

Type Conversions

Fully implemented

Qualified Expressions

Fully implemented

Quantified Expressions

Not implemented

Conditional Expressions

Fully implemented

Modifiers

Fully implemented

Definedness of Expressions

All expressions are assumed to be defined always.

C.6.5 Statement Annotations

Fully implemented

C.6.6 Annotation of Subprograms

All subprograms (actual and virtual) must be supplied with bodies. If there is a separate subprogram declaration and body, then their subprogram annotations must *conform* with each other, *even* if the subprogram body appears in a package body. In fact, the annotations of a subprogram declaration (with a separate body) are actually ignored.

C.6.7 Packages

Package Structure

Fully implemented

Visible Annotations in Package Specifications

Modified subtype annotations are not implemented.

Hidden Package Annotations

Fully implemented, unless specifically mentioned elsewhere.

Annotations on Private Types

Fully implemented

Package States

Not implemented

Axiomatic Annotations

Axiomatic annotations that fall into the subset as described in Chapter 4 are processed for semantic correctness. Checks are performed to ensure that these annotations do indeed meet the subset requirements. However, the package and the annotations are not transformed for runtime checking.

Consistency of Anna Packages

This section is *not implemented.* However, the only effect it has is to force the subprogram annotations in the subprogram specification and body to conform to each other as has already been mentioned earlier.

Example of a Package with Annotations

Irrelevant

C.6.8 Visibility Rules in Annotations

Fully implemented

C.6.9 Tasks

Irrelevant

C.6.10 Program Structure

Fully implemented

C.6.11 Exception Annotations

Both strong and weak propagation annotations are fully implemented. However, they are restricted to appear as the first set of annotations within any block. For this purpose, subprogram annotations are also considered part of the subprogram body and considered to appear earlier than any other annotation in the body. Propagation annotations that do not appear as required are ignored. The names of propagation annotations are ignored.

C.6.12 Annotation of Generic Units

Object annotations are not allowed in generic formal parts.

C.6.13 Annotation of Implementation-Dependent Features

Irrelevant

Bibliography

[1] Infotech International. *Infotech State of the Art Report, Software Testing Volume 1: Analysis and Bibliography*, 1979.

[2] *The Ada Programming Language Reference Manual.* US Department of Defense, US Government Printing Office, February 1983. ANSI/MIL-STD-1815A-1983.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques and Tools.* Addison-Wesley, 1986.

[5] S. Alagić and M. A. Arbib. *The Design of Well-Structured and Correct Programs. Texts and Monographs in Computer Science*, Springer-Verlag, 1978.

[6] A. L. Ambler, D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch, and R. E. Wells. GYPSY: a language for specification. *ACM SIGPLAN Notices*, 12(3):1–10, March 1977.

[7] L. M. Augustin, B. A. Gennart, Y. Huh, D. C. Luckham, and A. G. Stanculescu. Verification of VHDL designs using VAL. In *Proceedings of the 25th Design Automation Conference (DAC)*, pages 48–53, Anaheim, CA, June 1988.

[8] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[9] G. Birkhoff. On the structure of abstract algebras. *Proceedings of the Cambridge Philosophical Society*, 31:433–454, 1935.

[10] B. W. Boehm, R. K. McClean, and D. B. Urfrig. Some experiences with automated aids to the design of large-scale reliable software. In *Proceedings of the International Conference on Reliable Software*, pages 105–113, April 1975.

[11] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, April 1975.

[12] R. S. Boyer and J. S. Moore. Proving theorems about LISP functions. *Journal of the ACM*, 22(1):129–144, January 1975.

[13] M. Brooks. *Determining Correctness by Testing.* Technical Report 80-804, Department of Computer Science, Stanford University, May 1980.

[14] S. Burris and H. P. Sankappanavar. *A Course in Universal Algebra.* Springer-Verlag, 1981.

[15] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, second edition, 1984.

[16] N. H. Cohen. *Ada as a Second Language.* McGraw Hill, 1986.

[17] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, 1972.

[18] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Program mutation: a new approach to program testing. In *Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers*, pages 107–126, Infotech International, 1979.

[19] E. W. Dijkstra. *A Discipline of Programming. Series in Automatic Computation.* Prentice-Hall, 1976.

[20] A. Ersoz, D. M. Andrews, and E. J. McCluskey. *The Watchdog Task: Concurrent Error Detection Using Assertions.* Technical Report 85-267, Computer Systems Laboratory, Stanford University, May 1985.

[21] G. Estrin, D. Hopkins, B. Coggan, and S. D. Crocker. SNUPER COMPUTER—a computer in instrumentation automation. *AFIPS—Spring Joint Computer Conference*, 30:645–656, 1967.

[22] A. Evans, K. J. Butler, G. Goos, and W. A. Wulf. *DIANA Reference Manual, Revision 3*. Tartan Laboratories, Inc., Pittsburgh, PA, 1983.

[23] R. W. Floyd. Assigning meanings to programs. In *Proceedings of a Symposium in Applied Mathematics of the American Mathematical Society*, pages 19–32. American Mathematical Society, 1967.

[24] S. L. Gerhart and L. Yelowitz. Observations of fallibility in applications of modern programming methodologies. *IEEE Transactions on Software Engineering*, SE-2(3):195–207, September 1976.

[25] C. M. Geschke, J. H. Morris Jr., and E. Satterthwaite. Early experience with Mesa. *Communications of the ACM*, 20(8):540–553, August 1977.

[26] D. I. Good. Provable programming. In *Proceedings of the International Conference on Reliable Software*, pages 411–419, April 1975.

[27] D. I. Good, R. L. London, and W. W. Bledsoe. An interactive program verification system. *IEEE Transactions on Software Engineering*, SE-1(1):59–67, March 1975.

[28] D. I. Good and L. C. Ragland. Nucleus—a language for provable programs. In William C. Hetzel, editor, *Program Test Methods*, pages 93–117, Prentice-Hall, 1973.

[29] J. B. Goodenough and S. L. Gerhart. Towards a theory of test data selection. In *Proceedings of the International Conference on Reliable Software*, pages 493–510, April 1975.

[30] C. Green and D. R. Barstow. On program synthesis knowledge. *Artificial Intelligence*, 10:241–279, 1978.

[31] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich. *Report on a Knowledge Based Software Assistant*. Technical Report, Kestrel Institute, 1983.

[32] C. Green, J. Philips, S. Westfold, T. Pressburger, B. Kedzierski, S. Angebranndt, B. Mont-Reynaud, and S. Tappel. *Research on Knowledge-Based Programming and Algorithm Design*. Technical Report, Kestrel Institute, 1981.

[33] C. Green and S. Westfold. *Knowledge-Based Programming Self Applied*. Technical Report, Kestrel Institute, 1981.

[34] D. Gries. *The Science of Programming. Texts and Monographs in Computer Science*. Springer-Verlag, 1981.

[35] J. V. Guttag. The design of data type specifications. *Communications of the ACM*. 20(6):396–404, June 1977.

[36] J. V. Guttag. Notes on type abstraction (version 2). *IEEE Transactions on Software Engineering*, SE-6(1):13–23, January 1980.

[37] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[38] J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[39] J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, December 1978.

[40] J. V. Guttag, E. Horowitz, and R. Musser. The design of data type specifications. In R. T. Yeh, editor, *Current Trends in Programming Methodology, Volume 4—Data Structuring*, chapter 4, pages 60–79, Prentice-Hall, 1978.

[41] D. P. Helmbold. *The Meaning of TSL: An Abstract Implementation of TSL-1*. Technical Report CSL–TR–88–353, Computer Systems Laboratory, Stanford University, March 1988. Also published by Computer Information Sciences Board, UC Santa Cruz as UCSC-CRL–87–29.

[42] D. P. Helmbold and D. C. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985. (Also Stanford University Computer Systems Laboratory Technical Report No. 84-262).

[43] D. P. Helmbold and D. C. Luckham. *Runtime Detection and Description of Deadness Errors in Ada Tasking*. Technical Report 83-249, Computer Systems Laboratory, Stanford University, November 1983. (Program Analysis and Verification Group Report 22).

[44] D. P. Helmbold and D. C. Luckham. TSL: task sequencing language. In *Ada in Use: Proceedings of the Ada International Conference*, pages 255–274, Cambridge University Press, May 1985.

[45] W. C. Hetzel, editor. *Program Test Methods. Series in Automatic Computation.* Prentice-Hall, 1973.

[46] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–581, October 1969.

[47] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[48] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2(4):335–355, 1973.

[49] W. E. Howden. Algebraic program testing. *Acta Informatica*, 10:53–66, 1978.

[50] S. Igarashi, R. L. London, and D. C. Luckham. Automatic program verification I: a logical basis and its implementation. *Acta Informatica*, 4:145–182, 1975.

[51] D. H. H. Ingnalls. *FETE, A FORTRAN Execution Time Estimator.* Technical Report 71-20, Department of Computer Science, Stanford University, February 1971.

[52] K. Jensen and N. Wirth. *PASCAL—User Manual and Report.* Springer-Verlag, second edition, 1974.

[53] M. S. Johnson. A software debugging glossary. *ACM SIGPLAN Notices*, 17(2):53–70, February 1982.

[54] S. Katz and Z. Manna. Towards automatic debugging of programs. In *Proceedings of the International Conference on Reliable Software*, pages 143–155, April 1975.

[55] J. C. King. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software*, pages 228–233, April 1975.

[56] J. C. King. Proving programs to be correct. *IEEE Transactions on Computers*, C-20(11):1331–1336, November 1971.

[57] D. E. Knuth. *The Art of Computer Programming, Volume 1—Fundamental Algorithms.* Addison-Wesley, second edition, 1973.

[58] D. E. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1(2):105–133, April-June 1971.

[59] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1969.

[60] B. Krieg-Brückner. Consistency checking in Ada and Anna: a transformational approach. *Ada Letters*, 3(2):46–54, September-October 1983.

[61] B. Krieg-Brückner. Transformation of interface specifications. 1985. PROSPECTRA Study Note M.1.1.S1-SN-2.0.

[62] B. Krieg-Brückner and D. C. Luckham. Anna: towards a language for annotating Ada programs. *ACM SIGPLAN Notices*, 15(11):128–138, 1980.

[63] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. L. Popek. Report on the programming language Euclid. *ACM SIGPLAN Notices*, 12(2), February 1977.

[64] R. J. Lipton and F. G. Sayward. The status of research on program mutation. In *Digest for the Workshop on Software Testing and Test Documentation*, pages 355–373, Ft. Lauderdale, FL, 1978.

[65] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[66] B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–59, April 1974.

[67] B. Liskov and S. Zilles. Specification techniques for data abstraction. *IEEE Transactions on Software Engineering*, SE-1(1):7–19, March 1975.

[68] R. L. London. A view of program verification. In *Proceedings of the International Conference on Reliable Software*, pages 534–545, April 1975.

[69] D. J. Lu. Watchdog processors and structural integrity checking. *IEEE Transactions on Computers*, C-31(7):681–685, July 1982.

[70] D. C. Luckham. Programming with specifications: an introduction to Anna, a language for specifying Ada programs. 1987. Draft report, 450 pages.

[71] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. *Stanford Pascal Verifier User Manual.* Technical Report 79-731, Department of Computer Science, Stanford University, March 1979. (Program Analysis and Verification Group Report 11).

[72] D. C. Luckham, D. P. Helmbold, S. Meldal, D. L. Bryan, and M. A. Haberler. *TSL: Task Sequencing Language for Specifying Distributed Ada Systems: TSL-1.* Technical Report CSL-TR-87-334, Stanford University, July 1987. Program Analysis and Verification Group Report PAVG-34.

[73] D. C. Luckham and W. Mann. Methodology for using specification analysis to debug formal specifications. In preparation.

[74] D. C. Luckham, R. B. Neff, and D. S. Rosenblum. *An Environment for Ada Software Development Based on Formal Specification.* Technical Report 86-305, Computer Systems Laboratory, Stanford University, August 1986. (Program Analysis and Verification Group Report 31).

[75] D. C. Luckham, S. Sankar, and S. Takahashi. The methodology of formal specification and hierarchical debugging. (In preparation).

[76] D. C. Luckham, S. Sankar, and S. Takahashi. Two dimensional pinpointing: an application of formal specification to debugging packages. Forthcoming Technical Report.

[77] D. C. Luckham and N. Suzuki. Verification of array, record, and pointer operations in PASCAL. *ACM Transactions on Programming Languages and Systems*, 1(2):226–244, October 1979.

[78] D. C. Luckham and F. W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–23, March 1985.

[79] D. C. Luckham, F. W. von Henke, B. Krieg-Brückner, and O. Owe. *Anna—A Language for Annotating Ada Programs.* Springer-Verlag—Lecture Notes in Computer Science No. 260, July 1987. (Also Stanford University Computer Systems Laboratory Technical Report No. 84-261).

[80] A. Mahmood and E. J. McCluskey. *Concurrent Error Detection Using Watchdog Processors—A Survey.* Technical Report 85-266, Computer Systems Laboratory, Stanford University, June 1985.

[81] M. Mandal and S. Sankar. Concurrent runtime testing of anna annotations. In preparation.

[82] W. Mann. Representation of an anna subset in predicate logic for specification analysis. In preparation.

[83] Z. Manna and R. Waldinger. The logic of computer programming. *IEEE Transactions on Software Engineering*, SE-4(3):199–229, May 1978.

[84] A. A. Markov. Impossibility of certain algorithms in the theory of associative systems. *Doklady Akad. Nauk. SSSR*, 55:587–590, 1947. (Also appeared in 58:353–356).

[85] S. Meldal. A note on abstraction, in particular with respect to the axiomatics of access types. (In preparation).

[86] G. Mendal et al. The Anna-1 user guide and installation manual. Computer Systems Laboratory, Stanford University, Stanford, California - 94305. Jan. 1989. Available upon request. Forthcoming Technical Report.

[87] B. Meyer. Eiffel: Reusability and Reliability. In Will Tracz, editor, *Software Reuse: Emerging Technology*, IEEE Computer Society Press, 1988.

[88] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[89] G. J. Meyers. *The Art of Software Testing.* John Wiley & Sons, New York, 1979.

[90] A. Mili. Self-stabilizing programs: the fault-tolerant capability of self-checking programs. *IEEE Transactions on Computers*, C-31(7):685–689, July 1982. (Also see correspondence in *IEEE Transactions on Computers*, C-34(1):97–98, January 1985).

[91] D. R. Musser. Abstract data type specification in the AFFIRM system. *IEEE Transactions on Software Engineering*, SE-6(1):24–32, January 1980.

[92] P. Naur. Proof of algorithms by general snapshots. *BIT*, 6(4):310–316, 1966.

[93] R. Neff. Ada/Anna package specification analysis. Forthcoming PhD Thesis.

[94] S. S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[95] D. L. Parnas. The influence of software structure on reliability. In *Proceedings of the International Conference on Reliable Software*, pages 358–362, April 1975.

[96] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[97] E. L. Post. Recursive unsolvability of a problem of thue. *Journal of Symbolic Logic*, 11:1–11, 1947.

[98] T. W. Pratt. *Programming Languages - Design and Implementation*. Prentice-Hall, 1984.

[99] C. V. Ramamoorthy and S. F. Ho. Testing large software with automated software evaluation systems. *IEEE Transactions on Software Engineering*, SE-1(1):46–58, March 1975.

[100] C. V. Ramamoorthy, K. H. Kim, and W. T. Chen. Optimal placement of software monitors aiding systematic testing. *IEEE Transactions on Software Engineering*, SE-1(4):403–411, December 1975.

[101] B. Randell. System structure for fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[102] C. Rich. Formal representation of plans in the Programmer's Apprentice. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1044–1052, 1981.

[103] C. Rich and R. C. Waters. *Abstraction, Inspection and Debugging in Programming*. Technical Report AIM–634, MIT Artificial Intelligence Laboratory, June 1981.

[104] D. S. Rosenblum. A methodology for the design of Ada transformation tools in a DIANA environment. *IEEE Software*, 2(2):24–33, March 1985. (Also Stanford University Computer Systems Laboratory Technical Report No. 85-269).

[105] D. S. Rosenblum, S. Sankar, and D. C. Luckham. Concurrent runtime checking of annotated Ada programs. In *Proceedings of the 6th Conference on Foundations*

of Software Technology and Theoretical Computer Science, pages 10–35. Springer-Verlag—Lecture Notes in Computer Science No. 241, December 1986. (Also Stanford University Computer Systems Laboratory Technical Report No. 86-312).

[106] E. C. Russell and G. Estrin. Measurement based automatic analysis of FORTRAN programs. *AFIPS—Spring Joint Computer Conference*, 34:723–732, 1969.

[107] S. Sankar. Specification of history sensitive functions in Anna. (In preparation).

[108] S. Sankar and D. S. Rosenblum. *The Complete Transformation Methodology for Sequential Runtime Checking of an Anna Subset.* Technical Report 86-301, Computer Systems Laboratory, Stanford University, June 1986. (Program Analysis and Verification Group Report 30).

[109] S. Sankar, D. S. Rosenblum, and R. B. Neff. An implementation of Anna. In *Ada in Use: Proceedings of the Ada International Conference, Paris*, pages 285–296. Cambridge University Press, May 1985.

[110] D. G. Shapiro. *SNIFFER: A System that Understands Bugs.* Technical Report AIM-638, MIT Artificial Intelligence Laboratory, June 1981.

[111] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983. (An ACM Distinguished Dissertation, 1982).

[112] E. M. Soloway, B. Woolf, E. Rubin, and P. Barth. MENO-II: an intelligent tutoring system for novice programmers. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI)*, pages 975–977, 1981.

[113] L. G. Stucki and G. L. Foshee. New assertion concepts for self-metric software validation. In *Proceedings of the International Conference on Reliable Software*, pages 59–65, April 1975.

[114] N. Suzuki. Verifying programs by algebraic and logical reduction. ICRS, 1975.

[115] D. J. Taylor and J. P. Black. Principles of data structure error correction. *IEEE Transactions on Computers*, C-31(7):602–608, July 1982.

[116] A. Thue. Probleme üeber veränderungen von zeichenreihen nach gegebenen regeln. *Kra. Vidensk. Selsk. Skrifter. I. Mat. Nat. Kl.*, 10, 1914.

[117] F. W. von Henke and D. C. Luckham. A methodology for verifying programs. ICRS. 1975.

[118] F. W. von Henke, D. C. Luckham, B. Krieg-Brückner, and O. Owe. Semantic specification of Ada packages. In *Ada in Use: Proceedings of the Ada International Conference*, pages 185–196, Cambridge University Press, May 1985.

[119] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.

[120] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

[121] D. B. Wortman. On legality assertions in EUCLID. *IEEE Transactions on Software Engineering*, SE-5(4):359–367, July 1979.

[122] W. A. Wulf, R. L. London, and M. Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Transactions on Software Engineering*, SE-2(4):253–265, December 1976.

[123] S. S. Yau and R. C. Cheung. Design of self-checking software. In *Proceedings of the International Conference on Reliable Software*, pages 450–457, April 1975.

[124] R. T. Yeh, editor. *Current Trends in Programming Methodology, Volume 1—Software Specification and Design.* Prentice-Hall, Inc., 1977.

[125] R. T. Yeh, editor. *Current Trends in Programming Methodology, Volume 2—Program Validation.* Prentice-Hall, Inc., 1977.

[126] R. T. Yeh, editor. *Current Trends in Programming Methodology, Volume 4—Data Structuring.* Prentice-Hall, Inc., 1978.

[127] S. N. Zilles. *Algebraic Specification of Data Types.* Project MAC Progress Report 11, Massachusetts Institute of Technology, 1974.